DISEÑO E IMPLEMENTACIÓN DE UNA SOLUCIÓN MÓVIL PARA LA BÚSQUEDA DE OFERTAS DE VIDEOJUEGOS

DESIGN AND IMPLEMENTATION OF A MOBILE SOLUTION FOR SEARCHING VIDEO GAME OFFERS

JOSÉ MIGUEL SARASOLA OLABUENAGA

MÁSTER EN INGENIERÍA INFORMÁTICA. FACULTAD DE INFORMÁTICA
UNIVERSIDAD COMPLUTENSE DE MADRID

Curso académico: 2018-2019
Convocatoria: junio-julio 2019

Calificación: 8.8

Director:

Enrique Martín Martín

Resumen

Diseño e implementación de una solución móvil para la búsqueda de ofertas de videojuegos

El proyecto descrito a continuación pretende ofrecer una solución a nivel profesional para la búsqueda de ofertas de videojuegos desde un dispositivo móvil. Para ello se ha trabajado en los distintos componentes necesarios para dicho proceso: un servidor desde el que se registra la información, se generan las bases de datos pertinentes y se ofrece el acceso a la información de los precios u otros servicios de una forma sencilla y correctamente formateada; y la correspondiente aplicación móvil que permite consumir la información proporcionada por el servidor.

El servidor se ha escrito con el lenguaje de programación *Rust* creado por la fundación *Mozilla*, que ofrece un alto rendimiento comparable al de otros lenguajes de sistemas como C o C++, pero con una mayor seguridad en tiempo de compilación respecto a accesos de memoria y concurrencia [1] además de una serie de herramientas y funcionalidades que se pueden encontrar en otros lenguajes de programación modernos, como el uso de gestores de compilación o la inserción de dependencias de forma simple desde el fichero de descripción del proyecto. La aplicación móvil se ha desarrollado con el marco de desarrollo *Flutte*r creado por *Google*, que ofrece una solución multiplataforma de alto rendimiento y utiliza el lenguaje de programación *Dart*.

El servidor hace uso de una base de datos *PostgreSQL* alojada en un contenedor *Docker* y la plataforma de advertencias de problemas *Sentry*, que notificará al administrador del sistema de problemas en el servidor.

Toda la solución se distribuye bajo licencia *MIT* y busca la colaboración de otros usuarios para crecer, ofreciendo una sencilla base para que otros usuarios puedan añadir nuevas tiendas desde las que obtener precios de videojuegos a la plataforma así como nuevas funcionalidades.

Palabras clave

Aplicación móvil, Rust, Flutter, PostgreSQL, Sentry, Docker, Videojuegos, Compra, Online

Abstract

Design and implementation of a mobile solution for searching video game offers

The presented project pretends to offer a professional solution in order to search videogame deals from mobiles devices. To achieve this, we have worked on the different parts related into the process: The server that records the information, generates the databases, and accesses the videogames prices from third-party websites in a simple way; and the mobile application that allows final users to consume the data provided by the server.

The server has been written in the Rust programming language, created by the Mozilla Foundation, which offers high performance similar to other systems language such as C and C++, but with a better security in compile time regarding the accessto memory and concurrency [1] and providing other features that can be found on other modern programming languages. The mobile application has been developed with the Flutter framework, created by Google, which offers a high performance multiplatform solution and uses the Dart programming language.

The server uses a PostgreSQL database hosted on a Docker container and the issue notification platform Sentry, which will send a message to the system administrator whenever there is a problem with the server.

The whole solution is distributed under the MIT license and seeks from other users collaboration in order to grow, offering a simple to use base so other users can quickly add new stores to get videogame prices to the platform.

Keywords

Mobile applicaction, Rust, Flutter, PostgreSQL, Sentry, Docker, Videogames, Shopping, On-line

Índice general

Índice I

Agradecimientos V

1. Introducción 1
- 1.1. La compra de videojuegos por Internet . . . 1
- 1.2. Objetivos principales . . . 2
- 1.3. Objetivos secundarios . . . 2
- 1.4. Planificación y plan de trabajo . . . 3
  - 1.4.1. Organización de tareas . . . 3
  - 1.4.2. Hitos del proyecto . . . 4

2. Preliminares 7
- 2.1. Rust . . . 7
- 2.2. Flutter . . . 8
- 2.3. Dart . . . 8
- 2.4. Docker . . . 9
- 2.5. PostgreSQL . . . 9
- 2.6. Actix . . . 10
- 2.7. Diesel . . . 10
- 2.8. Sentry . . . 10
- 2.9. Índices invertidos . . . 11
- 2.10. Trie . . . 11

3. Preparación del entorno de desarrollo 13
- 3.1. Requisitos para el servidor . . . 13
  - 3.1.1. Rust . . . 13
  - 3.1.2. PostgreSQL y Docker . . . 14
  - 3.1.3. Diesel . . . 15
- 3.2. Requisitos para la aplicación móvil . . . 16
  - 3.2.1. Android Studio . . . 16
  - 3.2.2. Flutter . . . 17
  - 3.2.3. Compilación de APK para Android . . . 18
- 3.3. Recomendaciones para trabajar sobre el proyecto . . . 19
  - 3.3.1. Sistema operativo . . . 19
  - 3.3.2. Kits de desarrollo . . . 20
  - 3.3.3. Editor de código . . . 20

3.3.4. Extensiones utilizadas . . . 21
3.3.5. Rust (Extensión de Visual Studio Code) . . . 22
3.3.6. Compatibilidad de las extensiones con el equipo de trabajo . . . 22
3.3.7. Equipo de trabajo . . . 22

4. Arquitectura **24**
4.1. Servidor . . . 24
4.2. Aplicación móvil . . . 25
4.3. Sitios externos . . . 25

5. Servidor **27**
5.1. Requerimiento de una base de datos . . . 27
5.2. Preparación de la base de datos . . . 28
5.2.1. Implementación de la interface de ORM . . . 28
5.3. Inserción de información básica en la base de datos . . . 31
5.4. Creación de los pricers . . . 35
5.5. Proceso principal . . . 37
5.5.1. Arrancar el proceso del servidor . . . 37
5.5.2. Mensajes de error personalizados . . . 37
5.5.3. Integración con Sentry . . . 38
5.5.4. Creación de las bases de datos de búsquedas . . . 41
5.5.5. Obtención de precios . . . 43
5.5.6. Obtención de métricas desde HowLongToBeat . . . 43

6. Aplicación móvil **45**
6.1. Selección del nombre . . . 45
6.2. Dependencias utilizadas . . . 45
6.2.1. cached_network_image . . . 46
6.2.2. nima . . . 46
6.2.3. sqflite . . . 46
6.2.4. share . . . 46
6.2.5. shared_preferences . . . 46
6.2.6. url_launcher . . . 47
6.2.7. intl . . . 47
6.3. Diseño de la aplicación móvil . . . 47
6.3.1. Diseño de la aplicación general . . . 47
6.3.2. Diseño de los elementos contenedores de paquetes . . . 48
6.3.3. Información desde ProtonDB . . . 53
6.3.4. Temas adaptables y soporte para Android Q . . . 54
6.4. Búsquedas . . . 55
6.4.1. gameDB.json . . . 55
6.4.2. invertedIndex.json . . . 56
6.4.3. Carga de los ficheros en memoria . . . 56

6.4.4. Motor de búsquedas . . . 57
6.5. Trabajando de forma asíncrona . . . 58
6.6. Diseño del acceso a los datos . . . 59
6.6.1. Creación y uso de la caché . . . 60
6.7. Estado del soporte multiplataforma . . . 60
6.8. Internacionalización . . . 61

7. Uso y desplegado del sistema **64**
7.1. Uso y desplegado del servidor . . . 64
7.1.1. Instalación de Docker . . . 64
7.1.2. Creación del contenedor de PostgreSQL . . . 65
7.1.3. Configuración de los proyectos para el uso de la base de datos . . . 66
7.1.4. Aplicar los esquemas de la base de datos . . . 67
7.1.5. Inserción de entradas en la base de datos . . . 67
7.1.6. Arranque del servicio del servidor . . . 67
7.2. Desplegado del servidor con Docker Compose . . . 68
7.3. Compilación e instalación de la aplicación . . . 69
7.4. Uso de la aplicación móvil . . . 72
7.4.1. Acceso a los ajustes . . . 72
7.4.2. Búsqueda de paquetes . . . 72
7.4.3. Vista principal con paquetes agregados . . . 73

8. Despliegue en la nube y retroalimentación **78**
8.1. Preparación para el alpha testing . . . 78
8.1.1. Preparación de la imagen Docker . . . 79
8.1.2. Preparación de infraestructura con Docker Compose . . . 80
8.2. Retroalimentación . . . 82
8.2.1. Flujo de trabajo . . . 83
8.2.2. Compras directas desde la aplicación . . . 84
8.2.3. Mejorar transiciones entre vistas . . . 84

9. Conclusiones y desafíos **85**
9.1. Conclusiones . . . 85
9.1.1. Rust . . . 86
9.1.2. Flutter . . . 87
9.1.3. Docker . . . 87
9.1.4. Proyecto basado en Web-Scraping . . . 88
9.1.5. Proyecto en general . . . 89
9.2. Ideas descartadas . . . 89
9.2.1. Recortador de imágenes inteligente . . . 90
9.2.2. HashMaps concurrentes . . . 90
9.3. Desafíos . . . 90
9.3.1. Obtener una base de datos de videojuegos . . . 91

9.3.2. Crear un motor de búsquedas rápido y con sugerencias en tiempo real 91
9.3.3. Manejo de errores . . . 92
9.3.4. Facilidad a la hora de crear e insertar Pricers . . . 92

10. Trabajo futuro **93**
10.1. Notificaciones en tiempo real . . . 93
10.2. Optimización en la entrega de la base de datos . . . 94
10.3. Envío del HTML de las tiendas desde el dispositivo móvil . . . 94

11. Introduction **96**
11.1. Purchasing videogames on the Internet . . . 96
11.2. Main goals . . . 96
11.3. Secondary goals . . . 97
11.4. Planning and work-plan . . . 98
11.4.1. Task organization . . . 98
11.4.2. Schedule of events . . . 98

12. Conclusions and challenges **102**
12.1. Conclusions . . . 102
12.1.1. Rust . . . 103
12.1.2. Flutter . . . 104
12.1.3. Docker . . . 104
12.1.4. Project based on Web-Scraping . . . 105
12.1.5. Project in general . . . 105
12.2. Discarded ideas . . . 106
12.2.1. Smart image crop . . . 106
12.2.2. Concurrent HashMaps . . . 107
12.3. Challenges . . . 107
12.3.1. Obtaining a videogame database . . . 107
12.3.2. Creating a fast search engine with suggestions in real time . . . 108
12.3.3. Error handling . . . 108
12.3.4. Creating and adding pricers easily . . . 108

Bibliografía **114**

Agradecimientos

Lo primero, agradecer a mis padres y mi familia su incondicional ayuda siempre que lo he necesitado, por todos los recursos que han puesto a mi disposición para poder llegar a donde he llegado y por su infinita paciencia.

Después de mi familia, darle las gracias al director de este proyecto, Enrique. Tras una gran impresión cursando una asignatura suya durante el máster en ingeniería informática, donde enseñó herramientas y conocimientos que a día de hoy ya me han sido de gran ayuda, aceptó ser el tutor del proyecto y sin su incansable ayuda no estaría donde estoy. Gracias por toda la ayuda durante el desarrollo del proyecto, por guiarme y aconsejarme ante las dudas y por todo el tiempo que ha decidido dedicar a este proyecto.

Un gracias muy especial a Michael, por su infinita sabiduría y por motivarme a aprender y utilizar Rust. Sin él este proyecto no habría sido ni la mitad de emocionante de lo que ha sido al no haber usado este lenguaje de programación.

Por último, darle las gracias a los amigos que he podido hacer durante mi estancia en Madrid mientras he cursado este máster. Gracias a ellos por hacer que el tiempo que he pasado aquí haya sido inmejorable y por su increíble capacidad para estudiar, que me ha motivado cada día a seguir estudiando y trabajando para ser un mejor ingeniero.

Este trabajo va dedicado a todos vosotros.

Capítulo 1

Introducción

1.1. La compra de videojuegos por Internet

La compra de videojuegos en formato digital ha ido creciendo a gran velocidad durante los últimos años, con *Steam* superando su número de usuarios y compradores activos año tras año [2]. Con el fin de evitar el monopolio en el mercado de videojuegos digitales en el sector del PC, *Steam* permite a las editoras de estos distribuir claves a otros medios para después activarse dentro del propio *Steam* y tener acceso a la mismas funcionalidades que las que tiene el usuario que decide comprar directamente la licencia en *Steam*. Esto hace que el usuario pueda encontrar una gran cantidad de tiendas digitales compitiendo entre sí con distintos descuentos y promociones para exactamente el mismo producto: una licencia para un juego en *Steam*. Además de esto, distintas tiendas digitales pueden poner sus juegos con descuentos durante periodos distintos, haciendo la compra de un juego específico en un momento concreto más rentable en una tienda que en otra. Este proyecto pretende facilitar el proceso de encontrar un juego al mejor precio en un momento específico sin la necesidad de que el usuario tenga que navegar por decenas de páginas web comparando los precios manualmente. Todo el proyecto se encuentra disponible en `https://gitlab.com/ofertus`.

1.2. Objetivos principales

El objetivo de este proyecto es ofrecer a los posibles compradores la información relevante respecto a los precios de distintos videojuegos de la forma más sencilla posible.

El principal objetivo del proyecto es el de implementar una solución a nivel profesional en el que el usuario final pueda buscar fácilmente los distintos precios de los videojuegos que quiera entre las distintas tiendas online que ofrezcan dicho producto. En el sector web ya existen soluciones similares, como *IsThereAnyDeal* [3] que ofrece los distintos precios de un videojuego entre distintas webs. Con este proyecto al que nominaremos a partir de ahora como Ofertus, se pretende ofrecer una *API REST* que ofrezca esa misma información con un formato estandarizado y específico, habilitando así el desarrollo de distintas aplicaciones que permitan a posibles programadores terceros implementar soluciones que faciliten la búsqueda de precios a los usuarios.

Esta solución se implementará como el producto del servidor, pero a su vez se ha desarrollado una aplicación móvil que consumirá la información proporcionada por el servidor a forma de demostración de las capacidades de éste para la obtención de información correctamente formateada sobre los distintos videojuegos y paquetes que ofrece *Steam* en su tienda digital. En el caso de *Android*, *Google Play Store* tiene distintas aplicaciones de observación de precios, enfocadas en tiendas especificas o aplicaciones donde comunidades de usuarios comparten ofertas entre ellos, pero no hay una solución que ofrezca directamente la comparativa de precios entre videojuegos de forma automatizada entre distintas tiendas digitales y Ofertus puede ser la primera solución de éste tipo en el mercado móvil.

1.3. Objetivos secundarios

Los objetivos secundarios de este proyecto incluyen la gestión de un proyecto junto a una comunidad, además del aprendizaje de distintas herramientas y lenguajes de desarrollo recientemente lanzados.

Todo el proyecto de Ofertus se ha creado con la intención de publicarse cómo código

abierto, distribuido bajo licencia *MIT* [4] y con acceso a éste a través de la plataforma *GitLab* [5]. La intención del proyecto es la de encontrar a nuevos programadores que deseen colaborar mejorando la funcionalidad o añadiendo nuevas tiendas digitales a analizar desde el servicio. Se ha hecho especial hincapié en que el código sea fácil de leer y entender, así como en la creación de abstracciones y bibliotecas que faciliten la colaboración de terceros para añadir nuevas funcionalidades al proyecto.

Por otro lado, Ofertus trata de un proyecto centrado en un concepto relativamente sencillo, ello hace de este proyecto un lugar ideal para aprender y experimentar con nuevos lenguajes y herramientas de desarrollo sin requerir de complejos algoritmos o conocimientos altamente avanzados. *Rust* se ha ganado el favor de muchos programadores en el ámbito de lenguajes de sistemas y se ha colocado como el lenguaje de programación más querido por los programadores en la última encuesta de *Stack Overflow* [6], una conocida comunidad de programadores. *Flutter* a su vez se posiciona como una gran alternativa en el sector de soluciones móviles multiplataforma, con un enfoque distinto a su adversario *React Native*. Dado que *Flutter* es un producto mucho más joven y que no esta basado en *JavaScript*, se ha convertido en la alternativa móvil más interesante para experimentar.

1.4. Planificación y plan de trabajo

La planificación del proyecto se realizará en base al tiempo disponible para su desarrollo, fijando la fecha del 1 de mayo como límite para el añadido de nuevas funcionalidades. A partir de dicha fecha solo se trabajará en solución de problemas y en la respectiva documentación.

1.4.1. Organización de tareas

Dado que solo hay una persona trabajando en el desarrollo del proyecto, éste será el único encargado de todas las tareas requeridas por dicho proyecto.

1.4.2. Hitos del proyecto

Creación de un servicio que proporcione precios desde Steam

El primer paso fue realizar un servidor básico en *Rust* que devolviera en formato *JSON* la información básica sobre los precios de un paquete en *Steam*. Esta tarea se realizó entre el 21 de diciembre de 2018 y el 28 de diciembre del mismo año.

Desarrollar el primer prototipo de la aplicación

Una vez se tenía acceso a la información básica desde *Steam*, se empezó a trabajar en los primeros bocetos y en la implementación de una aplicación móvil con *Flutter*. La aplicación debía mostrar la información que proporcionaba el servidor con un diseño agradable visualmente. Durante esta etapa se iteró varias veces respecto al diseño final de la aplicación. Más adelante se explica el proceso evolutivo respecto al diseño de esta. Este proceso se realizó entre las fechas 29 de diciembre de 2018 y el 16 de enero de 2019.

Generar una base de datos de juegos para búsquedas

Tras realizar un primer prototipo donde la información de los paquetes sobre los que se debían mostrar los precios habían sido añadidos manualmente, se procedió a la creación de una base de datos de videojuegos que permitiera la posterior realización de un motor de búsquedas local dentro del dispositivo móvil. Se trabajó con distintas herramientas y utilizando distintos mecanismos hasta que finalmente se optó por el actual sistema basado en *web scraping* y se trabajó exclusivamente en él entre el 17 de enero de 2019 y el 10 de febrero de 2019. Sin embargo se sigue trabajando en él actualmente para solucionar problemas debido a los distintos casos de uso que hay en la web de *Steam*.

Motor de búsquedas

Una vez se obtuvo una base de datos en base a los videojuegos ofrecidos por la tienda digital de *Steam*, se trabajó en la realización de un motor de búsquedas local en el dispositivo móvil en base a esta información. Esto requirió de la investigación sobre estructuras de datos

con tal de tener una solución que ofreciera resultados de forma eficaz en dispositivos móviles. Este trabajo se realizó entre el 11 de febrero de 2019 y el 24 de febrero del mismo año.

Desarrollo de distintos pricers

El siguiente paso era realizar distintas bibliotecas propias que utilizaría el servidor para la búsqueda de precios de un paquete de *Steam* en distintas tiendas digitales. Este proceso requirió de distintas iteraciones y pruebas debido a las distintas situaciones en las que se puede encontrar un paquete en cada una de las tiendas digitales (con o sin oferta, venta normal o preventa, gratuito temporalmente...). Se han realizado cuatro *pricers* y se sigue trabajando en ellos cuando se encuentra algún fallo durante las fases de prueba de la aplicación. Junto a estos *pricers* se desarrolló la interfaz que permite a un programador tercero crear su propio *web scraper* y que éste sea fácil de añadir a los ya existentes en el servidor. Esta tarea se realizó entre el 25 de febrero de 2019 y el 2 de abril del mismo año.

Desarrollo de la aplicación móvil

El último paso en la etapa de desarrollo fue la finalización de la aplicación móvil. El desarrollo ha pasado ya por dos refactorizaciones de código para mantenerse dentro de los estándares de *Flutter* que han cambiado ligeramente con el tiempo. Se ha incluido ajustes con filtros para las búsquedas, animaciones y transiciones, una página con las licencias de las bibliotecas empleadas, el ajuste de un modo oscuro y algunas funcionalidades más que se explican en su correspondiente sección. El desarrollo de la aplicación se ha realizado entre el 3 de abril de 2019 y el 28 de abril de 2019.

Despliegue de la solución en la nube

El paso final se realizó entre el 29 de abril de 2019 y el 6 de mayo del mismo año desplegando la solución en la nube de *Google*. Este proceso incluye la generación y osquestración del servidor en forma de contenedores *Docker* y su despliegue en la plataforma de *Google Cloud*.

Pruebas con la aplicación y solución de errores

Una vez el servidor se encontraba disponible públicamente, se envió la aplicación a un grupo selecto de individuos para realizar pruebas y solucionar los problemas que fueran apareciendo, así como la inserción de alguna nueva funcionalidad o pequeños cambios en el diseño. Este proceso se realizó entre el 8 de mayo de 2019 y el 20 de mayo del mismo año, donde se etiqueta el proyecto como beta, y se finaliza el desarrollo de éste hasta haber completado su correspondiente documentación.

Capítulo 2

Preliminares

A continuación se describirán los distintos lenguajes de programación y herramientas de desarrollo que se han utilizado para la implementación de los productos del proyecto:

2.1. Rust

Rust [7] es el lenguaje de programación creado por la fundación *Mozilla* en el año 2010. Se creó con el propósito de realizar un nuevo lenguaje de programación que tuviera un alto rendimiento, una gran estabilidad y una serie de funcionalidades y características que lo conviertan en un lenguaje productivo. *Rust* es un lenguaje de programación con proceso de compilación con el que se puede trabajar en cualquier plataforma generando compilaciones para cualquiera de las plataformas soportadas. Rust es de código abierto y se distribuye bajo licencia *Apache 2.0* [8] y *MIT* [4].

Las ventajas de utilizar *Rust* residen en su eficiencia tanto en el aspecto de procesamiento de datos como de uso de memoria, sin requerimientos de tiempos de ejecución o recolectores de basura. *Rust* es un lenguaje fuertemente tipado con un sistema de propiedad de modelos que aseguran el funcionamiento de la aplicación respecto al acceso a la memoria y en el uso de procesamiento en paralelo en tiempo de compilación.

Rust tiene tras de sí una amplia comunidad, una gran documentación, mensajes de error mucho más comprensibles que otros lenguajes y una serie de herramientas tanto para compilar como para gestionar los paquetes, que lo convierten en uno de los lenguajes de

programación favoritos entre los programadores de sistemas actuales.

Mozilla se encuentra actualmente reescribiendo componentes de *Firefox* [9] con este lenguaje para poder balancear la carga de trabajo en múltiples núcleos de forma segura y mejorar así el rendimiento, encontrándose en producción a día de hoy el motor de procesado de CSS, *Stylo* [10].

2.2. Flutter

Flutter [11] es un marco de trabajo desarrollado por *Google* que alcanzó su primera versión estable en diciembre de 2018. *Flutter* trabaja sobre el lenguaje de programación *Dart*, también creado por *Google* que actualmente se encuentra en su segunda versión. *Flutter* ofrece una solución para escribir código multiplataforma con una ejecución nativa en los dispositivos basados en *iOS* y *Android*. A pesar de parecer un producto similar a *React Native* [12], *Flutter* permite la creación de componentes distintos a los ofrecidos por las bibliotecas del sistema operativo y mantiene su alto rendimiento utilizando aceleración por *GPU* para la interfaz gráfica.

Flutter es un marco de trabajo que contiene funcionalidades como la recarga rápida, que permite visualizar cambios en la interfaz visual de la aplicación en tiempo real según se edita el código, una interfaz de usuario expresiva y flexible con una alta cantidad de elementos gráficos para diseñar una interfaz de usuario a medida y con alto rendimiento en dispositivos móviles.

2.3. Dart

Dart [13] es un lenguaje de programación creado por *Google*. El objetivo de *Dart* es ser un lenguaje de programación orientado al *frontend* con optimizaciones en tiempo de compilación y con soporte multiplataforma, siendo compatible con distintas plataformas como ARM, x86 o incluso la transpilación a *JavaScript*. *Dart* es compatible con miles de paquetes o bibliotecas disponibles en su ecosistema, es gratuito y de código abierto. Se

distribuye bajo una licencia propia.

El objetivo de *Dart* es el de ser un lenguaje de programación productivo, siendo un lenguaje con optimizaciones en tiempo en compilación y enfocando su atención en el rendimiento y en el tiempo de arranque. Su sintaxis contiene elementos de otros lenguajes de programación más utilizados y cualquier programador con experiencia en lenguajes como *C++*, *C#* o *Java* pueden entender y empezar a escribir código con unos pocos días de estudio.

Dart tiene orientación reactiva, permitiendo gestionar fácilmente objetos con cortos periodos de vida (como los elementos gráficos) y soporta fácilmente operaciones asíncronas en la interfaz de usuario, lo que facilita la creación de interfaces visuales fluidas que no dependen del procesamiento de datos.

2.4. Docker

Docker [14] es una tecnología de contenedores con el propósito de poder desplegar aplicaciones en sistemas independientemente de la infraestructura seleccionada. Permite la gestión de servicios y el mantenimiento de las aplicaciones de forma más sencilla a una instalación tradicional. Por otro lado los contenedores se ejecutan aislados entre sí para poder lanzar varios servicios en una misma maquina asegurando la seguridad e integridad de los datos que estos procesan. Los nuevos servicios para alojar aplicaciones web están preparadas para el despliegue de contenedores *Docker* y es la primera opción a la hora de desplegar un servicio web a día de hoy.

2.5. PostgreSQL

PostgreSQL [15] es un sistema de base de datos objeto-relacional y de código abierto. Utiliza el lenguaje *SQL* para las operaciones, funciona en los principales sistemas operativos y cumple con las propiedades *ACID* [16]. Tiene soporte para múltiples tipos de documentos y cuenta con una variedad de herramientas creadas por la comunidad que respalda el proyecto

por ser de código abierto y distribuirse bajo la licencia *PostreSQL* [17], una licencia similar a *BSD* o *MIT*.

2.6. Actix

Actix [18] es un marco de trabajo para crear aplicaciones web. Los características principales es que puede funcionar con la versión estable de *Rust*, es sencillo de utilizar, tiene funcionalidades como *WebSockets* o *HTTP/2*, es extensible y es muy rápido. *Actix* contiene las herramientas necesarias para ofrecer un servidor web con acceso *API REST* y es compatible con servicios de terceros que se han implementado en este proyecto.

2.7. Diesel

Diesel [19] es un *ORM* y constructor de consultas para *Rust*. Sus características, similares a las de *Actix*, es que previene errores en tiempo de ejecución y es muy rápido. Es un marco de desarrollo sencillo con el que trabajar y sobre el que el desarrollador crea su propia capa de abstracción contra la base de datos. Al estar escrito en *Rust*, tiene las ventajas de la alta velocidad y fiabilidad por lo que se ha escogido este marco de trabajo para toda la interacción con la base de datos. Es compatible con *PostgreSQL* entre otros sistemas de bases de datos.

2.8. Sentry

Sentry [20] es un sistema de seguimiento de errores de código abierto que permite a los desarrolladores monitorizar y solucionar problemas en tiempo real. Ofrece una agradable interfaz visual desde la que gestionar los distintos errores que puedan aparecer en el servidor junto con toda la información sobre el suceso.

Sentry también ofrece métricas y el aviso en tiempo real por correo electrónico sobre problemas ocurridos en el servidor. También permite la gestión de grupos y proyectos pa-

ra poder organizar mejor la asignación de los problemas a solucionar entre los distintos miembros de un equipo.

Se ha optado por esta herramienta para tener una visión global más sencilla sobre los errores que puedan ocurrir en la web en vez de simplemente tratar de leer y entender los logs donde se pueden mezclar entre sí distintos problemas y elementos no relacionados.

2.9. Índices invertidos

Los índices invertidos [21] son una estructura de datos que indexa las entradas existentes a partir del contenido. Su principal uso es el de realizar búsquedas o consultas dentro de documentos o de textos completos.

En el caso de este proyecto se ha implementado un índice invertido a nivel de palabra que asocia cada palabra existente entre los distintos títulos existentes, con el correspondiente identificador de dicho título. Se puede observar un ejemplo visual de la estructura de datos en la figura 2.1.

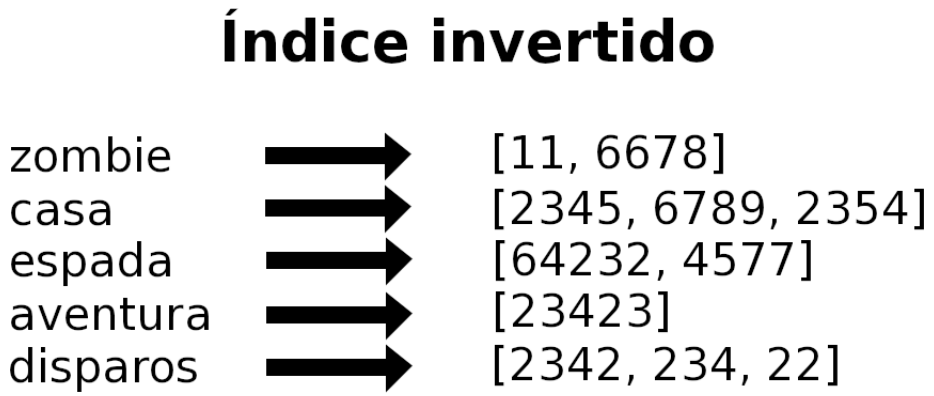

Figura 2.1: *Ejemplo de un índice invertido*

2.10. Trie

Un *Trie* [22] es un tipo de estructura de datos diseñada principalmente para la búsqueda de palabras. Para ello separa la palabra en letras y mantiene un árbol donde cada letra tiene

como nodo padre la letra que le precede en una palabra.

Aunque visualmente pueda resultar similar a un árbol tal y como se puede apreciar en la figura 2.2, en este caso el valor de un nodo es la concatenación de todos los nodos padres junto al nodo especificado. Para evitar problemas en los casos de uso de este proyecto, se ha asignado un valor a cada nodo indicando si se trata del final de una palabra o no, pero esto último no es obligatorio, sin embargo en algunos casos como en este proyecto es necesario para poder así indicar que, en el ejemplo de la figura, se encuentran las palabras: zebra, zombie y zombies.

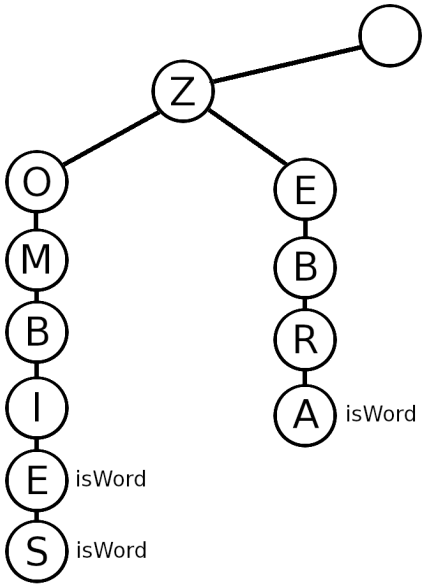

Figura 2.2: *Ejemplo de un árbol Trie*

Capítulo 3

Preparación del entorno de desarrollo

3.1. Requisitos para el servidor

Para poder trabajar y probar los proyectos relacionados con el servidor será necesario preparar los distintos *kits* de desarrollo de *software* respectivos a los lenguajes de programación y marcos de trabajo empleados. Además de esto, es altamente recomendable el uso de un entorno de desarrollo integrado o en su defecto un editor de código compatible con extensiones que agilicen trabajar sobre estas herramientas.

Todas las herramientas necesarias que se han utilizado durante el desarrollo se distribuyen de forma gratuita, son de código abierto y se encuentran disponibles para los tres sistemas operativos principales: *Windows*, *MacOS* y *Linux*. Durante el desarrollo se ha trabajado sobre *MacOS* y *Linux* por la comodidad que supone el desarrollo bajo plataformas *Unix-like*.

3.1.1. Rust

Rust es el lenguaje de programación utilizado para todo el servidor. Desde su página web, se puede instalar el asistente de instalación de *Rust*, *Rustup*.

Rustup

Rustup [23] ofrece una serie de utilidades sobre *Rust*, como la instalación de módulos para la compilación multiplataforma, el control de versiones del *kit* de desarrollo de *software*,

la posibilidad de cambiar entre versiones *Nightly*, *Beta* y Estable y la actualización de componentes a través de *CLI*.

Rustup además de instalar las herramientas para poder compilar *Rust*, también instala la herramienta de gestión de proyectos *Cargo* y los componentes necesarios para que los editores y código y los entornos de desarrollo integrado puedan ayudar a la hora de escribir código.

Cargo

Cargo [24] en la herramienta principal para gestionar los proyectos creados con *Rust*. Permite crear la plantilla básica de un proyecto escrito con *Rust*, la compilación y ejecución de código de forma sencilla, la ejecución de pruebas y el mantenimiento y la administración general de las dependencias de un proyecto. Es el equivalente a *NPM* [25] para el caso de proyectos que utilizan *NodeJS*.

Al tratar de compilar o ejecutar cualquiera de los proyectos creados con *Rust* utilizando *Cargo*, éste automáticamente descargará y compilará las dependencias necesarias en las versiones especificadas por los ficheros de configuración `Cargo.toml` y `Cargo.lock`, asegurando así que el código ejecutado en la máquina de cualquier usuario utiliza las mismas versiones que se han utilizado durante el desarrollo por parte del equipo de desarrollo.

3.1.2. PostgreSQL y Docker

Para tener un mejor control sobre la base de datos, incluyendo el uso explícito de una versión y la capacidad de llevarla a la nube en un futuro, se ha decidido utilizar una base de datos *PostgreSQL* a través del *software* de contención de *Docker*.

Docker

Docker permite la virtualización en forma de contenedores de prácticamente un sistema operativo *Unix-like* con todas las herramientas necesarias instaladas dentro de éste. Es la herramienta de contención número uno en el mundo y su uso no solo asegura una gran

facilidad para el despliegue en la nube de los servicios creados en la maquina local, si no que también permite el fácil mantenimiento de los servicios empleados dentro de este sin que estos afecten al sistema operativo anfitrión, manteniendo todos los ficheros de configuración y de uso del contenedor dentro de ficheros específicos de *Docker*, haciendo de la limpieza de estos archivos y configuraciones una tarea trivial.

El uso del contenedor de *PostgreSQL* es muy simple y tan solo se han configurado los ajustes de redirección de puertos y de volúmenes de persistencia para su uso durante el desarrollo del proyecto.

PostgreSQL

PostgreSQL se trata del mayor sistema de bases de datos relacional de código abierto disponible. Debido a su versatilidad, buena reputación y compatibilidad con sistemas *Linux* se ha optado por utilizar este sistema de bases de datos tras comprobar que se encuentra entre los soportados por el marco de trabajo de *Diesel*.

3.1.3. Diesel

Diesel es el marco de trabajo utilizado sobre el que se ha implementado la lógica *ORM* de la aplicación para poder así abstraer el servidor del funcionamiento interno de la base de datos que se decida utilizar. *Diesel* además de servir como biblioteca para estos propósitos también incluye una herramienta de *CLI* para realizar algunas gestiones como la preparación de las tablas de la base de datos antes de que el servidor trate de insertar datos en estas o el control de migraciones sobre la base de datos empleada.

La herramienta de *CLI* de *Diesel* se puede instalar a través de *Cargo*. *Rustup* durante la instalación inserta en las variables de entorno del sistema un directorio para binarios creados en *Rust*, permitiendo así utilizar aplicaciones creadas en *Rust* directamente en el sistema instalando estas a través de *Cargo*, sin necesidad de complejos procesos de compilación externos.

Al instalar la herramienta de *CLI* de *Diesel*, se descarga el código de la última versión

de esta y se compila directamente en la maquina, colocando los binarios ejecutables en el directorio de *Cargo* que *Rustup* ha agregado a las variables de entorno. Este proceso de compilación puede requerir de bibliotecas externas para el uso de distintos servidores de bases de datos.

Para el caso de *PostgreSQL*, se ha instalado la herramienta especificando que únicamente se requiere de compatibilidad con este servicio de base de datos a través del comando:

```
cargo install diesel_cli \
--no-default-features --features postgres
```

Sin embago, durante el proceso de compilación este requiere de las bibliotecas de desarrollo del sistema para PostgreSQL. En el caso del sistema operativo *Arch Linux*, este paquete se encuentra en los repositorios oficiales con el nombre `postgresql-libs` [26]. Es probable que se requieran de otras dependencias dentro del sistema operativo anfitrión de la maquina donde se deba compilar, como por ejemplo *CMake*.

3.2. Requisitos para la aplicación móvil

Para el desarrollo de la aplicación móvil son necesarias las herramientas de desarrollo para *Android* además de las de *Flutter*. Aunque el kit de desarrollo de software de *Android* se puede obtener por separado, el método recomendado para su instalación es a través del entorno de desarrollo integrado *Android Studio* que también es gratuito y compatible con los tres sistemas operativos principales.

3.2.1. Android Studio

Android Studio [27] contiene un asistente que permite instalar las herramientas del kit de desarrollo para *Android* necesarias así como los paquetes requeridos para crear una máquina virtual en el equipo. Sin embargo, tras este proceso será necesario la modificación de las variables de entorno del sistema para que herramientas externas a *Android Studio* puedan encontrar los binarios y las utilidades del kit de desarrollo. Más adelante se especifica cómo queda la configuración final de las variables de entorno para un equipo *Unix-like*.

3.2.2. Flutter

Finalmente se requiere del marco de trabajo *Flutter*, que en el caso de sistemas operativos *Unix-like*, se distribuye oficialmente como un directorio que se debe añadir a las variables de entorno del sistema.

Dado que Flutter precisa la creación de un fichero llamado `.flutter` en el directorio `$HOME`, se ha colocado el directorio del *SDK* de *Flutter* en `$HOME/.flutter-sdk`.

Antes de continuar, será necesario configurar las variables de entorno para poder ejecutar las herramientas de *Flutter* desde el terminal. A continuación se muestra la configuración utilizada en el equipo de desarrollo del proyecto:

```
# Rust Tools
export PATH="$HOME/.cargo/bin:$PATH"

# Android Tools
export ANDROID_HOME="$HOME/.android/sdk"
export PATH="$ANDROID_HOME/emulator:$ANDROID_HOME/tools:
$ANDROID_HOME/platform-tools:$PATH"

# Flutter Tools
export PATH="$HOME/.flutter-sdk/bin:$PATH"
```

Una vez esta configurado, se puede lanzar la herramienta de *CLI* como `flutter doctor`, el cual mostrará por pantalla el estado actual y posibles problemas con la instalación o la configuración del entorno. Si es la primera vez que se ejecuta *Flutter* en un sistema recién instalado, este avisará al usuario de que se deben aceptar varios términos y condiciones de la plataforma de *Android*, ofreciendo un asistente en la propia consola de comandos para realizar esta operación rápidamente.

El resultado final de lanzar `flutter doctor -v` debería ser similar al que se muestra en la figura 3.1 en equipos *Linux* donde no hay en ejecución ningún emulador de *Android* ni se encuentra ningún dispositivo *Android* conectado al equipo.

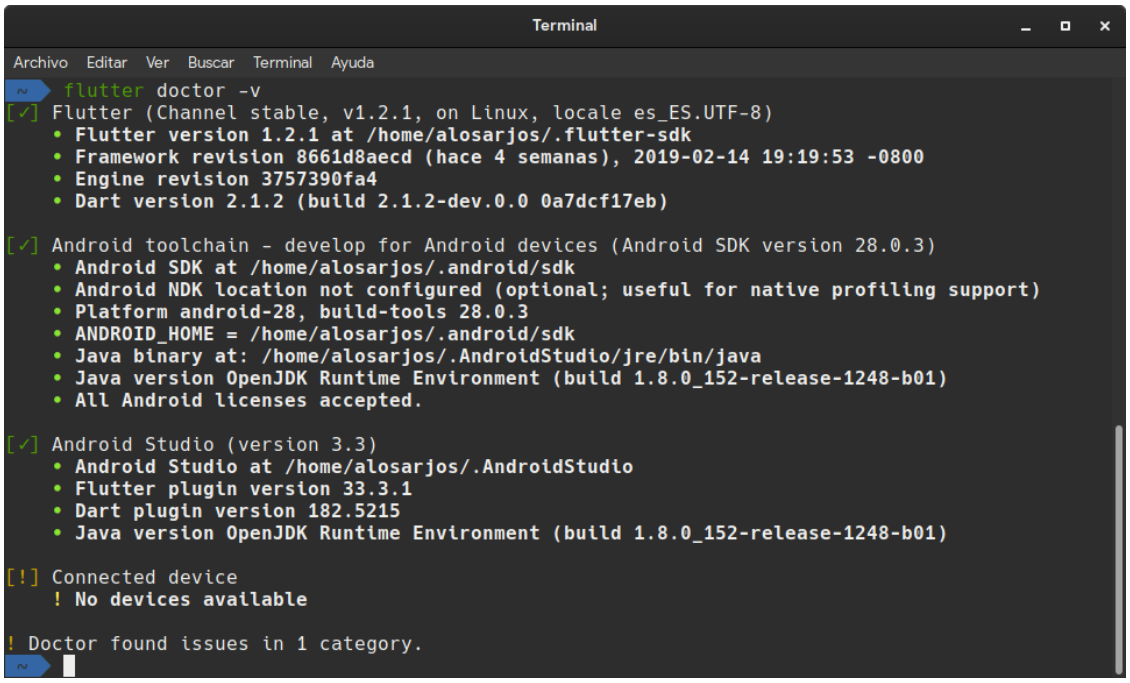

Figura 3.1: *Comprobación de estado de Flutter*

3.2.3. Compilación de APK para Android

De serie, el código de la aplicación para *Android* no compilará en equipos externos o ajenos al desarrollo. El uso de elementos como el inicio de sesión a través de las cuentas de *Google* activas en el teléfono o la capacidad de recibir notificaciones requieren de aplicaciones firmadas por el desarrollador y relacionadas con los correspondientes proyectos en la nube como el caso de *Firebase* (para el caso de las notificaciones).

Dado que estos ficheros deben mantenerse de forma privada por el equipo de desarrollo, no se distribuirán ni publicarán de forma abierta. Pero se agregarán los pasos necesarios para generar los ficheros correspondientes o si no estas funcionalidades pueden ser simplemente desactivadas de la aplicación y esta podrá ser compilada entonces.

El primer fichero importante es el listado de claves privadas. Este fichero se genera desde el asistente integrado en *Android Studio* y genera un fichero que contiene las claves privadas relacionadas con cada aplicación por parte del equipo de desarrollo o empresa. Mantener este fichero en secreto implica que las aplicaciones firmadas por él provienen de

dicho desarrollador asegurando así la integridad de la *APK* que el usuario final instalará.

Este fichero se conservará de forma privada y local en el equipo de desarrollo y se referenciará en el fichero de `key.properties` que se encuentra en el subdirectorio `android` del proyecto de *Flutter*. Este subdirectorio contiene todo el código y ficheros de configuración exclusivos de la plataforma de *Google* y solo deben ser modificados para ajustes avanzados de la aplicación, como es el caso de la firma de esta o de la ejecución de código en segundo plano, lo cual ha sido necesario durante el desarrollo. El fichero `key.properties` incluye una ruta hacia la localización del contenedor de claves así como las contraseñas requeridas para poder acceder a esta y que durante el proceso de compilación el compilador tenga acceso a la información necesaria para poder firmar la aplicación adecuadamente.

El proceso de firma durante la compilación es llevado a cabo por la herramienta *Gradle*. Es por ello que se requiere de una configuración adicional dentro del proyecto, más específicamente en el fichero `build.grade` que se encuentra en el directorio `android/app`. En este fichero se ha incluido un enlace al fichero `key.properties` para abstraer la configuración del acceso al listado de claves privadas de la lógica de compilación tanto como sea posible. Es por ello que un tercero tan solo requerirá generar un listado de llaves nuevo y colocar la información requerida dentro de un fichero `key.properties`.

3.3. Recomendaciones para trabajar sobre el proyecto

Aunque el proyecto puede ser leído y modificado en cualquier sistema operativo o editor de texto, se recomienda el uso de cierto entorno de desarrollo para evitar así problemas.

3.3.1. Sistema operativo

El sistema operativo recomendado simplemente debe tener base *Unix-like*, permitiendo así el fácil uso y configuración de herramientas *CLI*. Dado que las distribuciones *GNU/Linux* son las únicas que trabajan en la mayoría de sistemas y son gratuitas, se recomienda utilizar una distribución *GNU/Linux*. Las distribuciones utilizas durante el desarrollo han sido *Arch*

Linux [28] y *Ubuntu* [29]. Ambas distribuciones contienen en sus respectivos repositorios los paquetes necesarios para la instalación, configuración y funcionamiento del proyecto completo. A pesar de que otras distribuciones deberían también ofrecer los paquetes equivalentes a los encontrados en estas, el equipo de desarrollo solo puede garantizar la existencia de estos paquetes en versiones que funcionan adecuadamente con el proyecto en las 2 distribuciones previamente mencionadas.

3.3.2. Kits de desarrollo

Tanto en el caso de *Flutter* como de *Rust*, el desarrollo comenzó utilizando versiones *Nightly* o *Beta* de estas herramientas, pero finalmente, buscando una posible mayor estabilidad de cara al futuro se alteró el código para garantizar su funcionamiento sobre las versiones estables actuales de estos dos kits de desarrollo. Se garantiza la correcta compilación y funcionamiento utilizando *Flutter* en su versión `1.5.4` y *Rust* en su versión `1.33.0`. Durante la fase de desarrollo se ha procurado mantener el código actualizado y se han realizado múltiples modificaciones para aceptar los nuevos estándares y los cambios que han supuesto las diferentes actualizaciones de estas herramientas durante el tiempo de trabajo, pero es posible que las últimas versiones disponibles posteriores a la entrega de este proyecto (ya sea en el caso de las herramientas o de los *plugins* requeridos) sean incompatibles con el proyecto y éste requiera de nuevas modificaciones. Dado que *Flutter* es un kit de desarrollo con poco tiempo detrás y el proyecto basa parte de su funcionalidad en bibliotecas de terceros, la aplicación móvil tiene una mayor probabilidad de requerir modificaciones para poder mantener la compatibilidad con las últimas actualizaciones de las distintas bibliotecas empleadas.

3.3.3. Editor de código

Aunque cualquier editor de código es viable para trabajar sobre el proyecto, durante el desarrollo se ha optado por el uso de *Visual Studio Code* [30] de *Microsoft*. *Visual Studio Code* es un editor de texto orientado a código con un potente sistema de extensiones, de có-

digo abierto y que se distribuye gratuitamente. A pesar de estar desarrollado con tecnologías web, tiene un alto rendimiento y la compatibilidad con bibliotecas escritas en *JavaScript* permite que todas las funcionalidades y extensiones funcionen de igual manera entre los distintos tipos de sistemas operativos, no forzando así a los posibles colaboradores del proyecto a una plataforma especifica para trabajar.

3.3.4. Extensiones utilizadas

Dado que *Visual Studio Code* contiene soporte parta múltiples extensiones, se han empleado las recomendadas por los propios equipos de trabajo detrás de las herramientas empleadas, utilizando así las extensiones `Flutter` y `Rust (rls)`.

Flutter (Extensión de Visual Studio Code)

La extensión de *Flutter* para *Visual Studio Code* [31] principalmente cumple con tres tareas que agilizan en gran medida el desarrollo de proyectos creados con este *SDK*.

La primera es el colorear la sintaxis del código de forma que sea más sencillo leerlo. Dado que *Flutter* utiliza el lenguaje de programación *Dart*, la extensión *Dart* para *Visual Studio Code* [32] es instalada como dependencia de *Flutter*. Además de esto coloca comentarios en el editor de texto remarcando el final de los componentes. En un lenguaje como *Dart* donde se trabaja con un alta cantidad de componentes unos dentro de otros recursivamente, esta funcionalidad se vuelve realmente práctica, sobre todo a la hora de refactorizar código.

En segundo lugar la extensión de *Flutter* ofrece corrección de errores durante la edición de código sin tener que esperar al tiempo de compilación o de ejecución como es el caso de otros lenguajes de programación como *Python*.

En tercer y último lugar ofrece acceso a las herramientas del *CLI* de *Flutter* desde el propio *Visual Studio Code*. Esto incluye lanzar la aplicación en un dispositivo conectado o en un emulador y la posibilidad de ver en vivo en la aplicación los cambios que se realizan en el código. Esta última funcionalidad queda limitada al aspecto visual de la aplicación, puesto que el cambio de estados requiere de una secuencia de acciones o la interacción del

usuario, pero sigue siendo una funcionalidad muy útil durante el prototipado de la interfaz de usuario de la aplicación, agilizando mucho el tiempo necesario para realizar pruebas sobre los cambios realizados.

3.3.5. Rust (Extensión de Visual Studio Code)

La extensión de *Rust* para *Visual Studio Code* [33] hace uso del módulo *RLS* de Rust para ofrecer una serie de capacidades durante la escritura de código fuente en el editor. *RLS (Rust Language Server)* [34] es un servicio que se ejecuta de fondo que provee a servicios y aplicaciones de terceros como es en este caso *Visual Studio Code*, información sobre los programas escritos en *Rust*. Esto supone facilitar tareas como el renombramiento de variables, la refactorización, búsqueda de símbolos y el auto-completado de código.

3.3.6. Compatibilidad de las extensiones con el equipo de trabajo

Para que las extensiones sean capaces de encontrar los binarios contra los que trabajan, las variables de entorno deben ser configuradas en el intérprete de comandos por defecto del usuario. Si como es el caso del equipo de desarrollo actual, se utiliza un interprete de comandos distinto al por defecto, como puede ser *ZSH*, la configuración de las variables de entorno apuntando a los directorios que contienen los binarios deberá ser configurado en el fichero `$HOME/.zprofile`. De no ser así, el editor de código (que en este caso se trata de *Visual Studio Code*) deberá ser lanzado desde un terminal en el que previamente se hayan configurado las variables de entorno.

3.3.7. Equipo de trabajo

Aunque el equipo de trabajo es si cabe aún, más opcional que las herramientas mencionadas para trabajar sobre el proyecto, durante ciertas operaciones es recomendable tener ciertas características o funcionalidades que agilicen el proceso de trabajo sobre el proyecto.

Durante la tarea de *web-scraping* que recolecta la información sobre los distintos paquetes disponibles en la plataforma de *Steam*, es altamente recomendable tener un sistema

con un microprocesador con arquitectura multinúcleo. El proceso de *web-scraping* ha sido optimizado para aprovechar este tipo de arquitectura y lanzará varios procesos en paralelo para cada uno de los hilos de ejecución que la maquina disponga salvo que el usuario indique manualmente el número de hilos con los que desea trabajar. Dado que cada hilo realiza una petición web, será recomendable tener un ancho de banda proporcional al número de hilos de ejecución en paralelo con los que se vaya a lanzar el programa de recopilación de paquetes. Con un equipo utilizando 16 hilos de procesamiento en paralelo se recomienda un ancho de banda de 1 MB/s para evitar retrasos en la funcionalidad de los distintos hilos de ejecución. Destacar que la ejecución en paralelo acelera en gran medida el procesamiento de datos y disminuye de forma considerable el tiempo de ejecución necesario para la obtención e inserción de los paquetes disponibles en *Steam* dentro del sistema de base de datos.

Capítulo 4

Arquitectura

En este capítulo se explicará la arquitectura general del proyecto, realizando una explicación de alto nivel que permita comprender las interacciones entre los distintos componentes realizados.

Tal y como se puede observar en la figura 4.1, podemos agrupar los distintos elementos empleados para el funcionamiento del proyecto en tres conjuntos: servidor, aplicación móvil y sitios web externos. El servidor es el principal responsable de la captura de información y generación de bases de datos, la aplicación móvil es el método preferido para el consumo de dicha información y los sitios externos proveen precios o información extra para complementar la información mostrada en la aplicación.

4.1. Servidor

Lo primero que se puede observar en el servidor es que todos los procesos se ejecutan dentro de un contenedor *Docker* que a su vez se encuentra desplegado en *Google Cloud*. `steam web-scraper` se encarga de conectarse a *Steam* para la obtención de los paquetes disponibles en el catálogo a través de técnicas de *web-scraping* para posteriormente añadirlas dentro de una base de datos *PostgreSQL*.

Por otro lado, el proceso del servidor se encarga de generar un par de ficheros *JSON* requeridos por la aplicación móvil en base a la información de la base de datos, siendo estos generados cuando el `steam web-scraper` finaliza su tarea diaría de actualizar la base de

datos de videojuegos. Estos ficheros *JSON* son enviados a la aplicación móvil cuando esta lo solicite para poder hacer uso del motor de búsquedas local que se ha implementado en esta. Además de esto, el servidor también se encarga de ejecutar los *pricers*, encargados de realizar *web-scraping* en distintos sitios externos para obtener los precios actualizados, aquí representados por la propia *Steam*, *Humble Bundle* y *2Game*.

Finalmente, el servidor también obtiene información desde *HowLongToBeat* para la obtención de la duración de los videojuegos. Cómo esta tarea también requiere de *web-scraping*, se ha delegado al servidor para reducir el rendimiento requerido en los dispositivos móviles.

4.2. Aplicación móvil

La aplicación móvil por su parte solicita los ficheros *JSON* para poder realizar búsquedas de paquetes de forma local al servidor y los guarda dentro del directorio de ficheros especificado por *Android* o *iOS* para esta. Los paquetes seleccionados se añaden a una lista de seguimiento que se guarda de forma persistente en una base de datos *SQLite* para poder recuperarlos en el caso de que la aplicación se cierre. La aplicación se encargará de solicitar los precios al servidor, haciendo que este comience el procesamiento con los *pricers*. Por otro lado, la aplicación móvil obtendrá la información de *ProtonDB* directamente desde la *API REST* que tiene disponible. Dado que en el caso de *ProtonDB* no hay que hacer apenas procesamiento de la información, esta tarea se realiza directamente desde la aplicación móvil.

4.3. Sitios externos

Para la obtención de los precios de los sitios externos, se realizan téncicas de *web-scraping* en la mayoría de estos, puesto que ninguno ofrece una *API REST* para poder acceder a la información rápidamente. En algunos casos donde la web funciona a través de *JavaScript*, se puede recopilar cierta información como los resultados de búsquedas utilizando algunas *API REST* que la tienda utiliza internamente para dibujar dinámicamente el listado de

videojuegos, pero en la mayoría de las ocasiones el trabajo se reduce a investigar cómo funciona la tienda web utilizando las herramientas de desarrollo de *Firefox* y obtener la información de la mejor manera posible. Sin embargo hay varios sitios externos de los que que se ha tratado de obtener la información pero no ha sido posible por sus rebuscados métodos para obtener la información. Un ejemplo de estos casos sería *Fanatical* [35], donde cada sesión de nevagación obtiene una *API Key* temporal que se utiliza para realizar las consultas de forma asíncrona. Esto implica que la ejecución de código *JavaScript* es necesario y por lo tanto no se puede obtener la información por mecanismos de *web-scraping* convencionales.

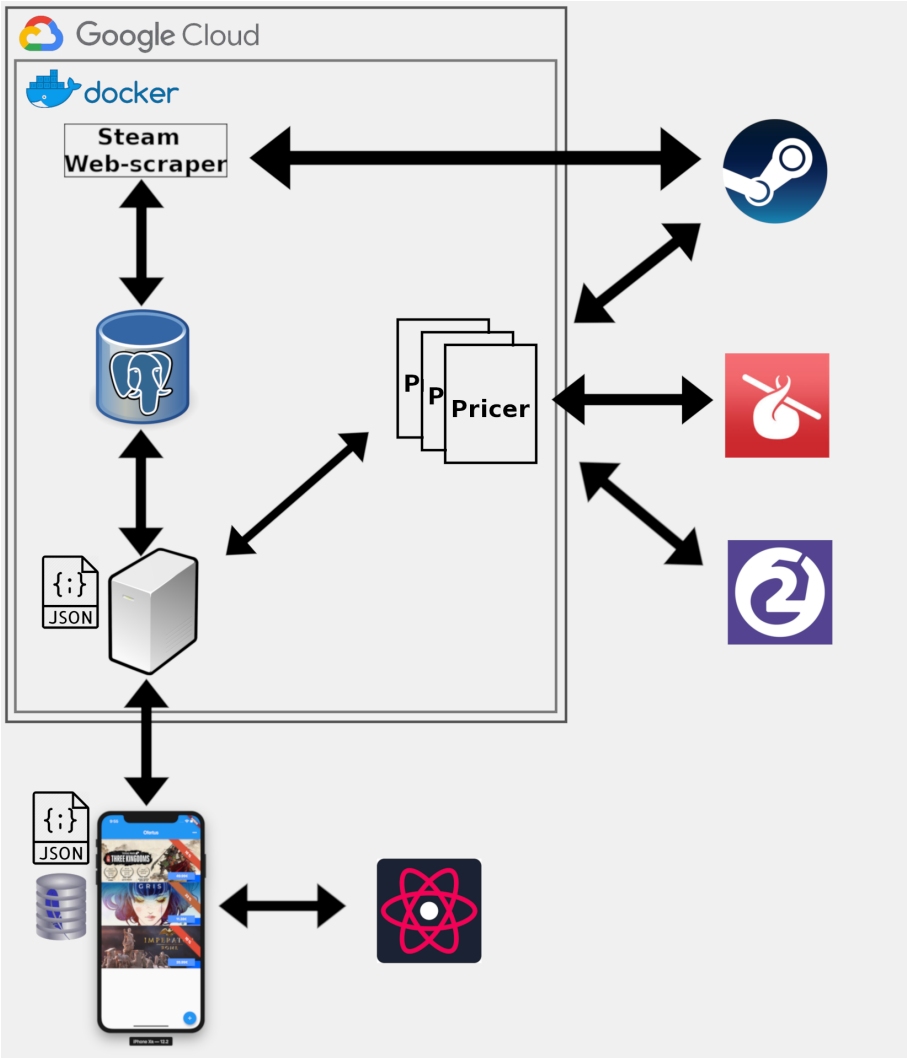

Figura 4.1: *Arquitectura del proyecto*

Capítulo 5

Servidor

5.1. Requerimiento de una base de datos

Dado que el proyecto se basa en la búsqueda de información y en el acceso a esta de la forma más rápida posible, se requiere de una base de datos desde la que poder obtener toda la información necesaria tanto para la búsqueda de precios en servicios de terceros así como el acceso al listado de videojuegos existentes sobre los que poder buscar los mejores precios desde la aplicación móvil.

Para la base de datos se ha escogido utilizar *PostgreSQL*, el cual ofrece una base de datos relacional de alto rendimiento con licencia gratuita y además proporciona una alta compatibilidad con las herramientas de trabajo que se han escogido utilizar para este proyecto.

Durante el desarrollo se ha trabajado con una base de datos *PostgreSQL* funcionando dentro de un contenedor *Docker*, lo cual permite aislar el proceso y mantener l servicio de gestión de bases de datos independientemente del sistema anfitrión. Esto permite que en el momento que se necesite realizar un despliegue, el hecho de poder configurar un contenedor *Docker* suponga una gran compatibilidad ofrecida por los servicios en la nube actuales.

El servidor requiere de una base de datos donde se insertará la información relevante a cada paquete. Esta base de datos se genera con el propósito de poder generar ficheros que la aplicación móvil cliente necesita para realizar búsquedas y para ofrecer tanta cantidad de

información como sea posible a los *pricers* (Las bibliotecas creadas para obtener precios de distintos sitios web. Se explican con mayor detalle más adelante).

Dado que este proyecto utiliza *Steam* como la fuente de información principal respecto al listado de videojuegos existentes para buscar, se requiere de una forma con la que poder acceder a dicho listado de forma rápida. *Steam* no ofrece ningún tipo de acceso a desarrolladores a este tipo de información por lo que se requiere de un acceso a un servicio de terceros o realizar una base de datos propia para este proyecto.

Como servicio de terceros, la única alternativa es *SteamDB* [36] que precisamente cuenta con una base de datos sobre todos los paquetes que contiene *Steam* así como bastante información sobre cada uno de estos. Sin embargo no ofrece ningún tipo de acceso por *API* tampoco y en sus normas queda estrictamente prohibido el uso de *Web Scrapers* para la obtención de la información.

Debido a estos factores se ha optado por la implementación de una base de datos propia en la que almacenar todo el listado de paquetes ofrecidos mediante *web-scraping* desde *Steam*.

5.2. Preparación de la base de datos

Para poder preparar la base de datos se ha utilizado el marco de trabajo de *Diesel*, una herramienta sobre la que se pueden desarrollar *interfaces ORM* compatibles con el lenguaje de programación *Rust*.

Diesel ofrece un marco de trabajo y algunas herramientas para realizar una implementación propia de una interfaz de *ORM* para poder realizar operaciones contra una base de datos de forma más programática en *Rust*.

5.2.1. Implementación de la interface de ORM

La implementación de la interfaz de *ORM* se encuentra dentro del subproyecto llamado `diesel` [19]. Este proyecto cuenta con los modelos y las migraciones requeridas para que el

proyecto pueda trabajar contra una base de datos *PostgreSQL*.

En primer lugar se encuentran las migraciones, desde las que se realizan los distintos pasos durante la preparación de la base de datos. Estas migraciones contienen la lógica de creación y eliminación de tablas según se realizan las migraciones en el orden en el que se han creado. Esto permite preparar la base de datos hasta un punto específico que no tiene por qué ser el final en las instrucciones de preparación de las tablas u otro contenido permitiendo incluso el volver hacia atrás eliminando en el orden preciso tablas y contenidos. El concepto de migración es común entre otros sistemas de bases de datos como por ejemplo *Microsoft SQL Server* [37]. Un ejemplo sencillo de migración es la creación y destrucción de tablas, tal y como se puede ver en los ejemplos 5.1 y 5.2

Listado 5.1: *Fichero up.rs*

```
CREATE TABLE packages (
  id SERIAL PRIMARY KEY,
  title VARCHAR NOT NULL,
  package_type VARCHAR NOT NULL,
  bundle_id VARCHAR NOT NULL,
  is_win BOOLEAN NOT NULL,
  is_mac BOOLEAN NOT NULL,
  is_linux BOOLEAN NOT NULL
)
```

Listado 5.2: *Fichero down.rs*

```
DROP table packages
```

Por otro lado se encuentran los modelos, que contienen la lógica referente a las operaciones de cada estructura y que se guarda dentro de la base de datos. Esto permite abstraer toda la lógica *SQL* dentro de la biblioteca haciendo así que el código base del servidor sea independiente del tipo de base de datos utiliza. Finalmente se encuentra el código base de preparación y el esquema final.

El código base de preparación cuenta con simples instrucciones para conectarse a la base de datos utilizando el fichero de configuración de entorno que especifica la *URL* en la que se encuentra la base de datos. Este fichero también contiene el usuario y la contraseña de la base de datos para que la configuración sea lo más simple posible. El esquema final se trata de un fichero de código que *Diesel* crea en base a las tablas creadas en los ficheros de

configuración de las migraciones y por lo tanto modificarlo no sirve para nada puesto que este fichero se genera automáticamente.

Una vez la *interface* de base de datos ha sido preparada, se utiliza la herramienta de *CLI* de *Diesel* para conectarse a la base de datos y crear las tablas y el contenido dentro de *PostgreSQL* según la especificación dentro del proyecto.

Este subproyecto a su vez se utiliza como biblioteca dentro de los subproyectos que lo requieran para tener acceso a los distintos objetos que se encuentran dentro de la base de datos y sus correspondientes operaciones.

Uso de la herramienta CLI de Diesel

La herramienta *CLI* de *Diesel* permite realizar uan serie de interacciones contra la base de datos de forma conveniente. Para este proyecto se han utilizado las más comunes.

Una vez que la herramienta de *CLI* se ha instalado, será necesario configurar la dirección en la que el sistema de base de datos se encuentra para que la herramienta de *CLI* pueda acceder a esta y realizar operaciones. Para un ajuste rápido, es suficiente con configurar la variable de entorno `DATABASE_URL`, como por ejemplo:

```
export DATABASE_URL=
postgres://username:password@localhost/diesel_demo
```

Pero en un buen entorno de desarrollo, este tipo de parámetros se configuran dentro de los ficheros del proyecto, más específicamente dentro del fichero `.env` que se encuentra dentro de cada proyecto que requiera de acceso a la base de datos.

Una vez configurada la variable de entorno `DATABASE_URL` o con el terminal situado en el directorio donde se encuentra el fichero `.env` (que contiene la información de las variables de entorno sin necesidad de configurar variables de entorno), con lanzar `diesel setup`, la herramienta generará la base de datos (en caso de que esta no exista) y creará un directorio vacío de migraciones donde se configurará el esquema.

El siguiente paso es la creación de las migraciones. Las migraciones son los distintos pasos o etapas de configuración en una base de datos y suelen incluir las instrucciones necesarias

para realizar las operaciones asignadas a la migración y las operaciones necesarias para deshacer el trabajo hecho. Para ello se utiliza el comando

```
diesel migration generate create_games
```

Se generan un par de ficheros por cada migración, `up.sql` y `down.sql` para las operaciones ha realizar y las operaciones para deshacer respectivamente. Dentro del fichero `up.sql` en este caso solo se genera la tabla de la base de datos que contiene la información de los paquetes.

```
  CREATE TABLE packages (
  id SERIAL PRIMARY KEY,
  title VARCHAR NOT NULL,
  package_type VARCHAR NOT NULL,
  bundle_id VARCHAR NOT NULL,
  is_win BOOLEAN NOT NULL,
  is_mac BOOLEAN NOT NULL,
  is_linux BOOLEAN NOT NULL
)
```

Y dentro del fichero de `down.sql`, la operación para la eliminación de la tabla y de sus datos.

```
    DROP table packages
```

Con estos ficheros configurados, se puede lanzar la herramienta de *Diesel* para ejecutar la migración en la base de datos

```
    diesel migration run
```

O para deshacer la última migración en la base de datos.

```
    diesel migration redo
```

A partir de este punto todo lo que se requiere hacer con la herramienta es aplicar y deshacer migraciones y ejecutar el comando `setup` en el caso de tener que reiniciar la base de datos.

5.3. Inserción de información básica en la base de datos

El proyecto requiere de una base de datos de paquetes sobre los que poder realizar consultas y analizar la información. Como fuente de obtención de videojuegos existentes se ha utilizado la plataforma *Steam*, que es la actual tienda de videojuegos más grande

existente. Sin embargo, debido a que Steam no ofrece una *API* o un mecanismo de obtención de todo el catálogo de juegos, se ha optado por crear un programa cuyo cometido es realizar *Web-Scraping* sobre *Steam*.

Esta utilidad se ha creado como una herramienta de *CLI* en la que se pueden designar manualmente el número de hilos de ejecución en paralelo que se desean utilizar, usando por defecto tantos como permita el anfitrión.

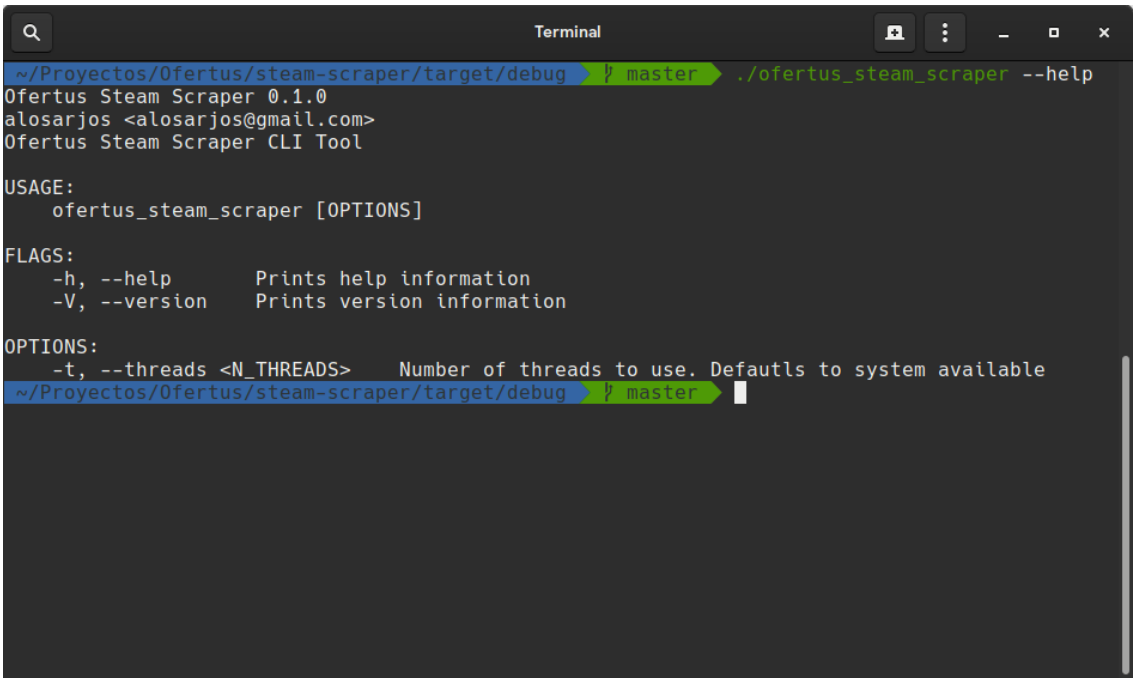

Figura 5.1: *Parámetros de ejecución del scraper de Steam*

Este programa realiza una búsqueda vacía en *Steam* simplemente aplicando los filtros de contenido que se desean obtener, que en el caso del proyecto se ha limitado a juegos, contenido descargable para estos y *bundles* de cualquiera de estos dos. Se realiza esta primera búsqueda para obtener la cantidad de paginas que devuelve *Steam*, donde cada página contiene un máximo de veinticinco entradas. Para este proceso se utilizan dos marcos de trabajo de *Rust*: *Reqwest* y *Scraper*.

Al obtener el número de páginas que se requieren pasar por el *Web-Scraper*, se avanza a través de estas con un tipo de iterador especial que permite la ejecución en paralelo de

código, ejecutando en hilos independientes el código relacionado a cada iterador, que en este caso se trata de cada página de *Steam* sobre la que se debe obtener la información. El procesamiento de estas páginas de forma paralela se ha hecho con el uso de *Rayon*, una biblioteca de *Rust* que permite la fácil ejecución en paralelo de código dentro de iteradores.

Además *Rayon* permite controlar manualmente el número de hilos con los que este programa trabajará, aunque por defecto la aplicación utilizará tantos como la maquina anfitrión pueda ofrecerle, resultando así en una aplicación escalable dependiendo del entorno en el que se ejecute.

Dentro de cada hilo se realizará una nueva petición de la página correspondiente, y se utilizan técnicas de *Web-Scraping* como selectores *CSS* y expresiones regulares para obtener los datos necesarios del código HTML como el que se puede ver en la figura 5.2. Primero se comprobará no obstante si el identificador del juego ya se encuentra en la base de datos para evitar procesamiento de información innecesario. Una vez se ha obtenido toda la información, esta se agrega a la base de datos a través de la biblioteca `ofertus_server_diesel`. Los datos a introducir por paquete son:

- Título del paquetes: El título completo del paquete
- Tipo de paquete: Se determina si se trata de un videojuego o un paquete o conjunto de varios.
- Identificador del bundle: Identificador necesario para poder mostrar la imagen de cabecera correspondiente en caso de que el paquete se trate de un bundle.
- Sistemas operativos compatibles: Los sistemas operativos sobre los que el juego ofrece soporte.

Este proceso deberá ejecutarse cada vez que se necesite añadir los nuevos títulos lanzados en *Steam* desde la última que se ejecutó. Por ello se necesita que sea un proceso rápido de lanzar. Aunque es posible ordenar los títulos de forma descendente por fecha de lanzamiento

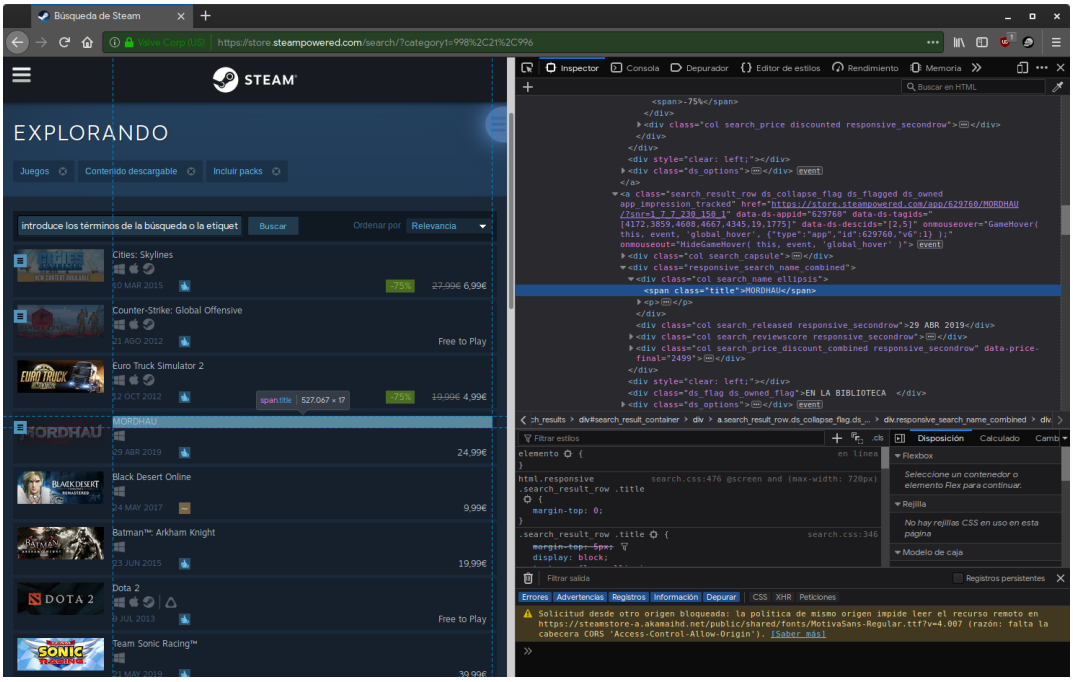

Figura 5.2: *Código fuente de Steam para el web scraper*

para así poder terminar antes el proceso, es posible que haya paquetes que pueden requerir ser actualizados por, por ejemplo, cambios en las plataformas soportadas, por ello se debe repetir el proceso para todas las entradas de paquetes en la web de *Steam*.

Una vez finalizado este proceso, se envía una señal a través de *Unix Sockets* al proceso principal del servidor para que éste regenere los ficheros *JSON* que requiere la aplicación móvil en base a los nuevos datos insertados en la base de datos de paquetes de *Steam*. El *Socket* se realiza a través de un fichero encontrado en `/tmp/ofertus_socket`.

El traspaso de código iterativo en un único hilo al multihilo en una máquina con dieciséis hilos de trabajo disponibles reduce muy significativamente el tiempo necesario para el procesamiento de la información, aunque la conexión a Internet y la estabilidad de los servidores de Steam son dos factores no controlables también importantes a la hora de analizar el rendimiento. Se muestran los resultados de la ejecución del proceso con 1 hilo de ejecución en la figura 5.3 y con 16 hilos en paralelo en la figura 5.4, que permite apreciar un tiempo de ejecución de 34:33 minutos en el primer caso y 2:08 minutos en el segundo.

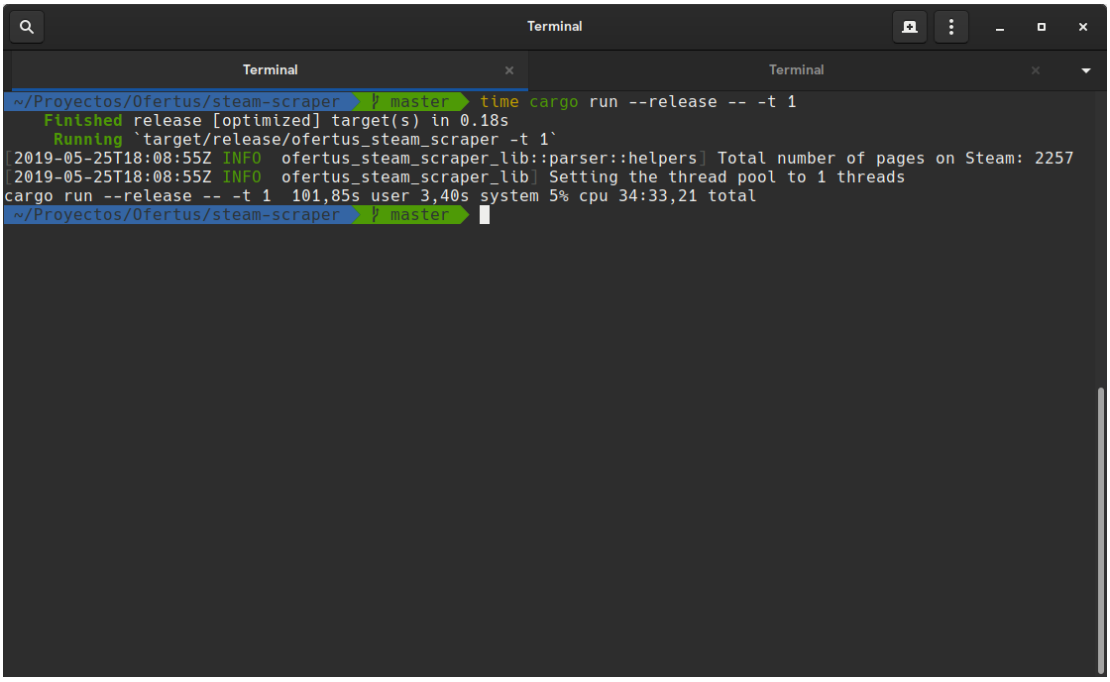

Figura 5.3: *Web scraper de Steam con 1 hilo de ejecución*

5.4. Creación de los pricers

Pricer es el nombre por el que se agrupan los *Web-Scrapers* orientados a la recogida de precios en distintas webs.

Para que voluntarios puedan agregar las tiendas que ellos consideren fácilmente, se ha implementado una interfaz simple, para que uno pueda crear un *pricer* por su cuenta y que luego su inserción en el servidor sea lo más simple posible.

Para desarrollar un *pricer*, el desarrollador voluntario debería crear una librería de *Rust*. Para ello puede utilizar la herramienta `cargo` de la siguiente manera donde `my_pricer` será el nombre que le quiera dar a la librería:

```
cargo new my_pricer --lib
```

Tras ello, deberá editar el fichero `lib.rs`, creando una estructura con el nombre del *pricer* e implementar la interfaz `Pricer` para dicha estructura. Esta interfaz tan solo requiere de la implementación de un método llamado `get_price_info`. Cómo implementar este método queda libre para el desarrollador.

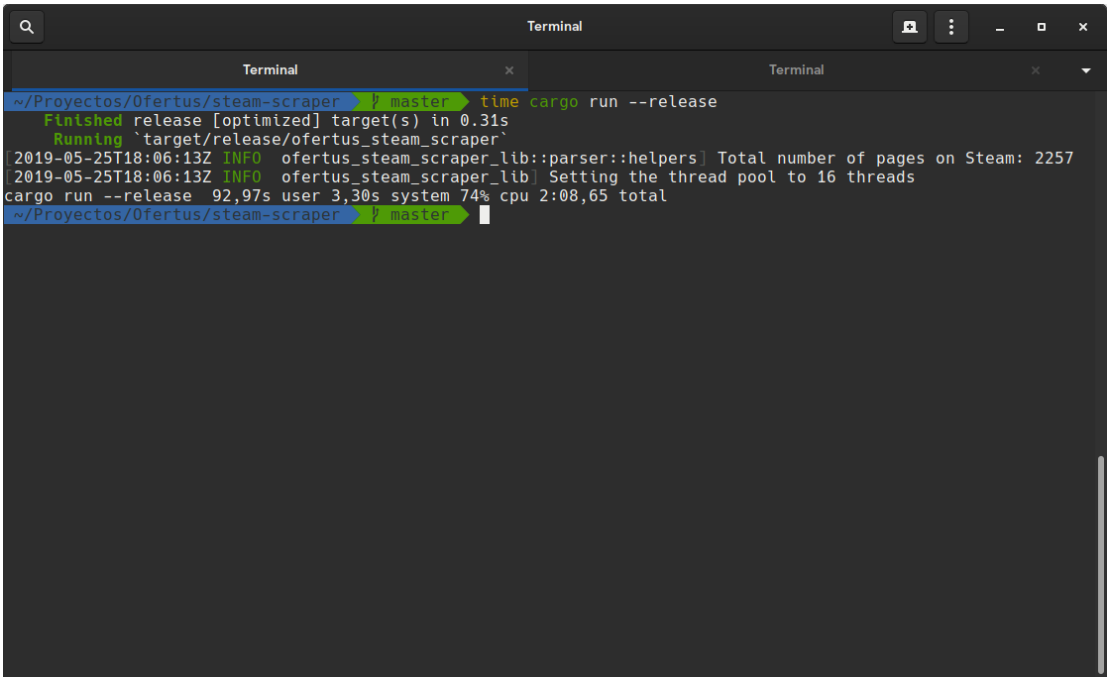

Figura 5.4: *Web scraper de Steam con 16 hilos de ejecución*

Éste método recibe un objeto del tipo `Package` y con ello toda la información que la base de datos tiene sobre el videojuegos que se debe buscar en la tienda y devolverá un `PriceInfo` envuelto en un `Option`.

El uso del `Option` en *Rust* permite trabajar sin necesidad de comprobaciones contra valores nulos, lo que evitar en gran medida el acceso incorrecto a memoria y posibles errores en tiempos de ejecución. Por otro lado el `PriceInfo` es una estructura de datos que contiene el precio actual, el precio regular, la *URL* del paquete en la correspondiente tienda, el descuento actual y el nombre de la tienda para que esta aparezca en la aplicación.

Los distintos *pricers* implementados utilizan este mecanismo por lo que el *fork* de uno de los *pricers* ya existentes debería ser una forma rápida de empezar a hacer uno propio, incluyendo el código referente a las pruebas para poder comprobar que funciona sin necesidad de tener que compilar y ejecutar toda la infraestructura del servidor en su ordenador personal.

5.5. Proceso principal

El proceso principal del servidor consta de de una inicialización del marco de trabajo *Actix* al que se le asocian las *URIs* que el servidor va a utilizar, las funciones asociadas a dichas *URIs* y el puerto en el que el proceso quedará a la escucha para las peticiones *HTTP*. Dado que el servidor se distribuirá como un contenedor de *Docker*, el puerto se deberá redireccionar al deseado en el exterior del contenedor y por lo tanto no es relevante el aquí asignado más allá de fines de desarrollo o depuración. El servidor también es el encargado de crear dos archivos *JSON* que la aplicación móvil descarga a modo de base de datos para realizar las búsquedas de forma eficiente en forma local.

El proceso principal lanza un hilo de ejecución en segundo plano que quedará a la escucha de mensajes de `steam-scraper` a través de un `Unix-Socket`.

5.5.1. Arrancar el proceso del servidor

El proceso del servidor se ha encapsulado como un programa configurable a través de *CLI* como se puede ver en la figura 5.5. Esto permite seleccionar el número de hilos de ejecución en paralelo máximo que puede utilizar así como el puerto a través del cual escuchará por peticiones *HTTP*. Por defecto escuchará en el puerto 8080 y utilizará tantos hilos como permita la maquina anfitrión del proceso.

5.5.2. Mensajes de error personalizados

Se ha implementado un sistema de errores personalizados que permite saber de forma más clara qué ha ocurrido en el servidor cuando este ha encontrado un problema de forma más clara. Este sistema se utiliza como una enumeración con mensajes personalizados y permite devolver errores en los distintos métodos del servidor de forma sencilla.

Gracias a este mecanismo se puede detectar rápidamente qué error ha ocurrido y en qué sección del código ha aparecido sin necesidad de leer la pila de ejecución del servidor en el registro de errores.

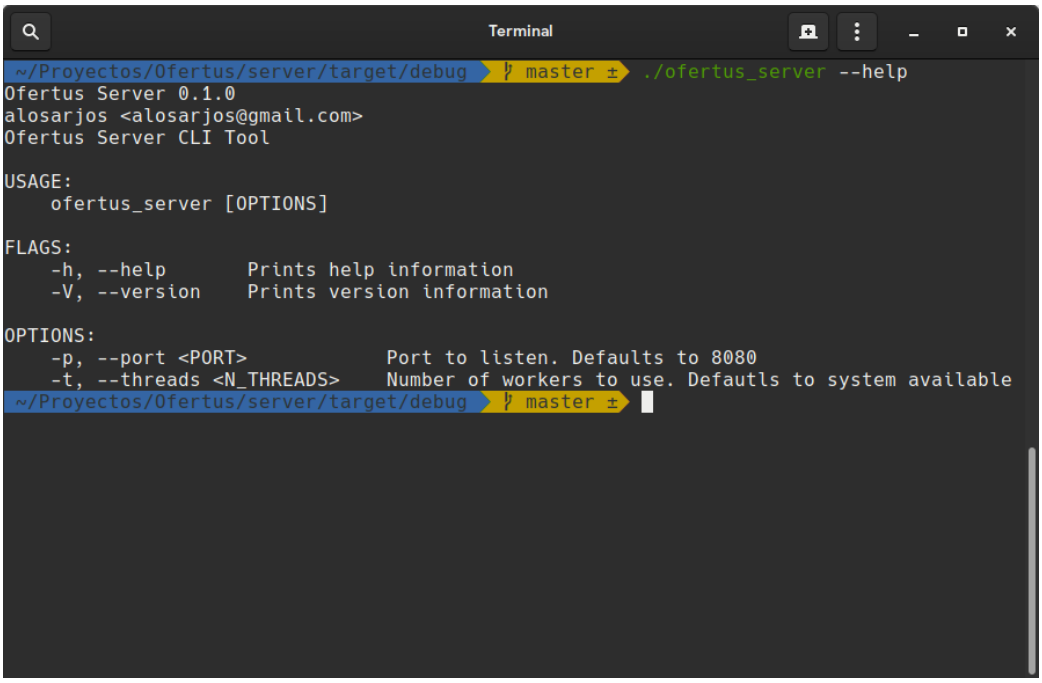

Figura 5.5: *Parámetros de ejecución del proceso del servidor*

5.5.3. Integración con Sentry

Sentry es un servicio de terceros que permite la gestión de errores de un servidor desde una interfaz de usuario a través de la cual es fácil navegar y operar.

Actix y *Sentry* han colaborado en una biblioteca que se inyecta como *middleware* en el proceso principal del servidor para que éste envíe los errores a *Sentry* además de a la salida de errores en el equipo donde se está ejecutando el proceso del servidor. Para activar este *middleware*, se ha registrado la integración con un manejador de mensajes de error tras inicializar el *middleware* con una clave privada ofrecida desde la web de *Sentry*.

El proceso principal del servidor trata de obtener la clave privada de comunicación de *Sentry* desde la variable de entorno de `SENTRY_DSN` y en el caso de encontrarla inicializa el servicio. Tanto si se activa como si no la conexión con *Sentry*, el servidor sigue mostrando todos los mensajes de error a través del sistema de *logs* propio.

Las ventajas que ofrece la integración con *Sentry* son el aviso de problemas en tiempo real a través de correo electrónico y un panel de información con los errores ocurridos en el

servidor de forma más cómoda a un *log* de texto.

En el caso de que ocurra un error se recibe un correo electrónico con la información con el aspecto que se puede ver en la figura 5.6. Accediendo al panel de información de *Sentry* se puede obtener más información como la que se puede ver en la figura 5.7.

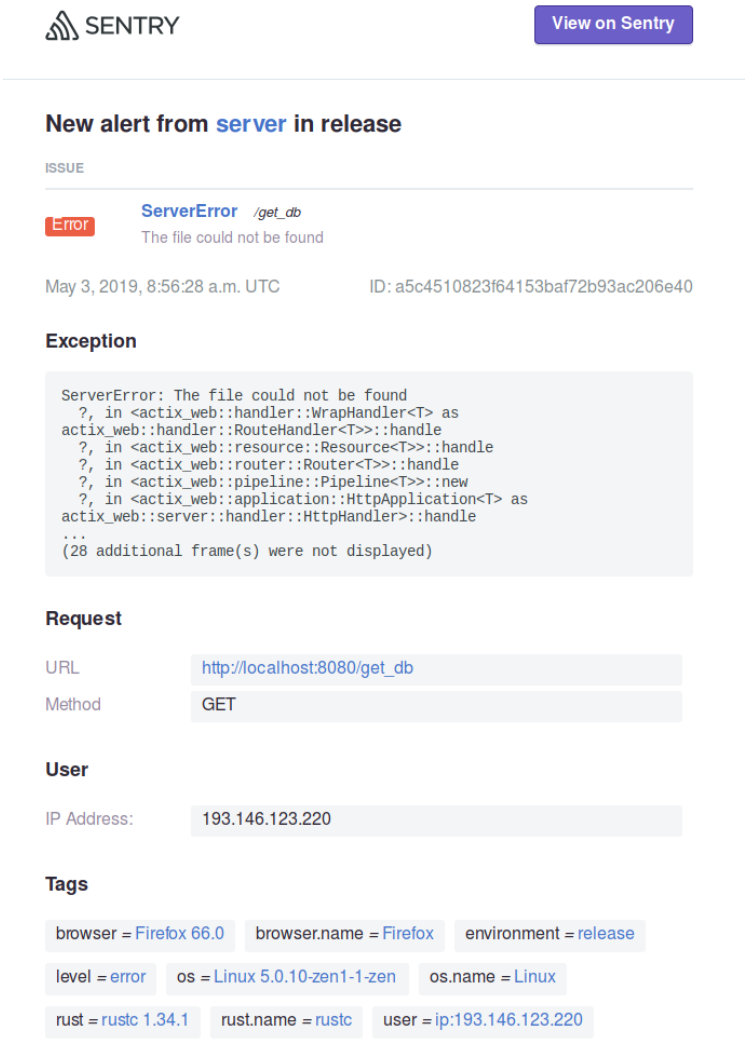

Figura 5.6: *Correo electrónico con el aviso de Sentry*

Accediendo a los detalles del error se obtiene una larga lista de información respecto al evento y al cliente que ha ocasionado el error en el servidor como el navegador web que utilizó para la petición o el sistema operativo que utiliza. Esto junto al sistema de errores personalizados que se ha implementado dentro del servidor permite saber rápidamente en

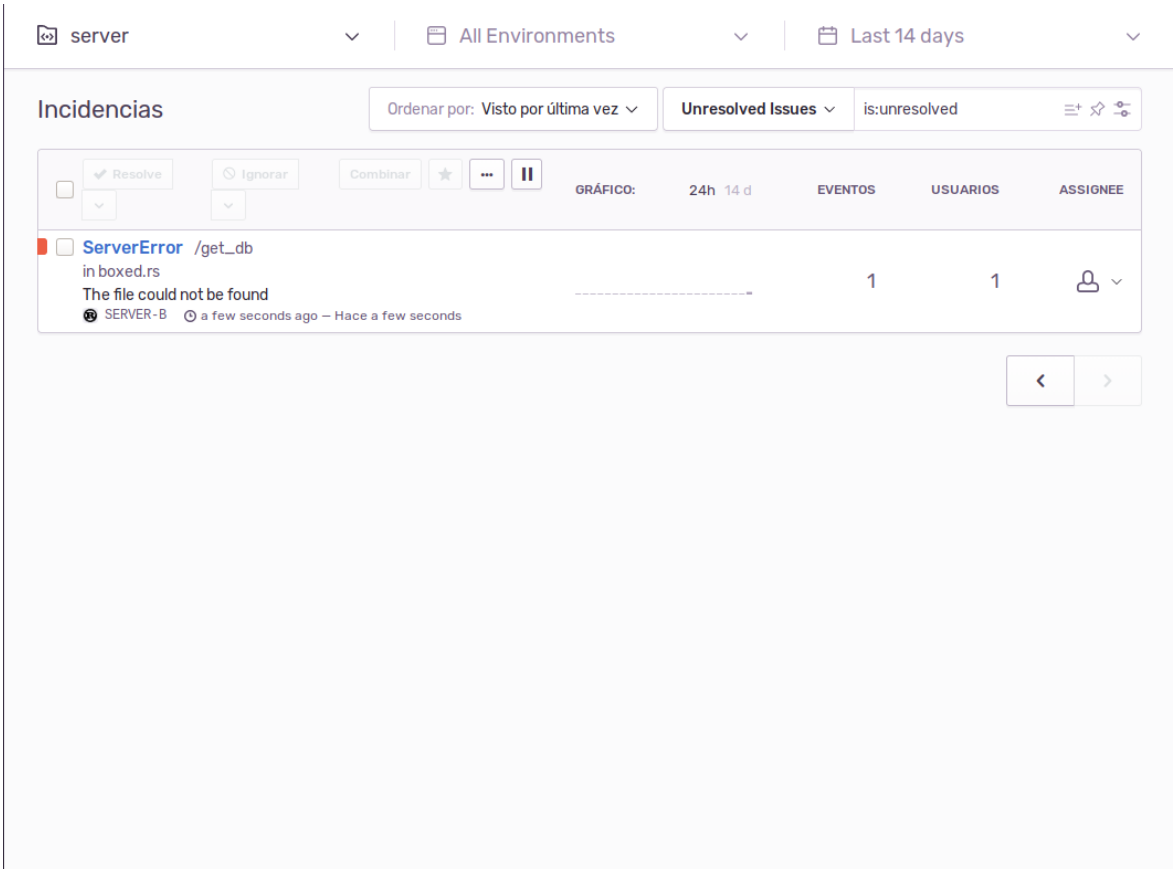

Figura 5.7: *Panel de control de Sentry para Ofertus*

qué parte del código ha ocurrido el error. En el caso de la figura 5.8 se puede observar que ha ocurrido un error con la descripción de *The file could not be found* al intentar acceder a la *URL* `get_db` por lo que se sabe que el fichero con la base de datos en formato *JSON* que debería existir, no lo hace.

Aún con el sistema de errores personalizados que se ha implementado en el servidor, Sentry sigue incluyendo una copia del *Stacktrace* en el caso de que el error personalizado no sea lo suficientemente descriptivo.

El uso de notificaciones a través de correo electrónico a un equipo de personas en tiempo real agiliza en gran medida la capacidad de respuesta en caso de encontrar problemas y es la principal funcionalidad por la que se ha decidido integrar *Sentry* en el proyecto.

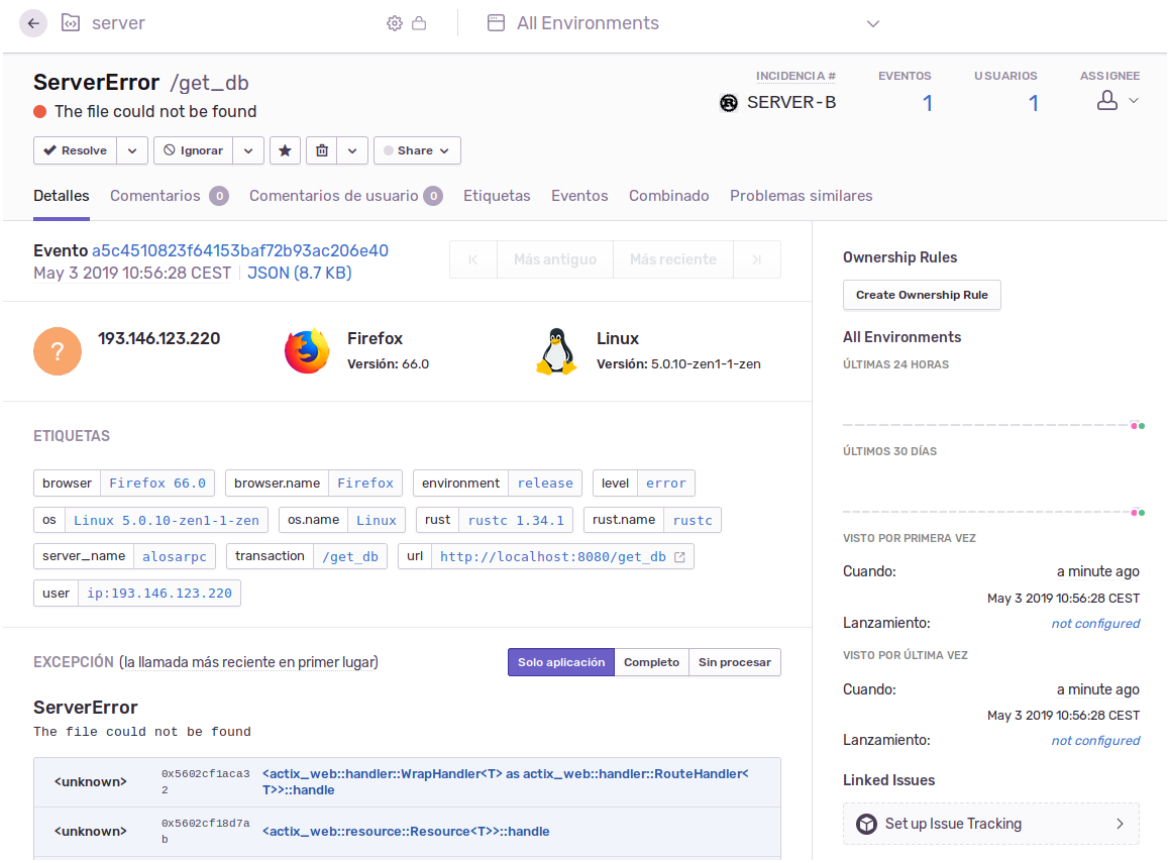

Figura 5.8: *Detalles del error en Sentry*

5.5.4. Creación de las bases de datos de búsquedas

La aplicación móvil requirió de la implementación de un motor de búsqueda. Para dicha implementación se requiere de un índice que incluya todos los juegos y la información conocida acerca de estos y un índice invertido con las palabras de los títulos para optimizar el tiempo de búsqueda dentro de la aplicación y reducir así la carga del teléfono y el consumo de batería de este.

Para ello el servidor genera estos dos bancos de información como dos ficheros *JSON* que permite su fácil revisión y depuración durante la fase de desarrollo. El uso de estos ficheros se analizará más a fondo en la correspondiente sección del motor de búsquedas dentro de la implementación de la aplicación móvil.

Para la generación de los archivos *JSON*, se recogen todas las entradas de la base de

datos y se organizan como un par de *HashMaps*, cuya serialización a través de la biblioteca de *Serde* facilitará la conversión a texto en formato *JSON*. La automatización del proceso de conversión a través de una biblioteca de serialización asegura que los *JSON* generados sean ficheros válidos.

Para generar el índice invertido, se limpian de los títulos los valores no alfanuméricos y se separa el título por palabras. En el caso de que la palabra exista como clave dentro del índice invertido, se le añadirá el identificador del paquete a la lista de valores que contiene esa clave, si la clave no existe se crea y se le asigna una lista con el nuevo identificador.

Base de datos de paquetes

La base de datos de paquetes contiene el identificador de los distintos paquetes dentro de la propia plataforma de *Steam* que se utiliza como fuente principal de paquetes en forma de índice. El resto de la información que contiene esta base de datos incluye:

- Título del paquete: El título completo del paquete
- Tipo de paquete: Se determina si se trata de un videojuego o un paquete o conjunto de varios.
- Identificador del *bundle*: Identificador necesario para poder mostrar la imagen de cabecera correspondiente en caso de que el paquete se trate de un bundle.
- Sistemas operativos compatibles: Los sistemas operativos sobre los que el juego ofrece soporte.

Obtención de los ficheros JSON

El servidor incluye dos *URIs* (`http://server_ip/get_db` y `http://server_p/get_ii`) que simplemente devuelven los ficheros *JSON* pre-generados como texto plano para que la aplicación pueda almacenar el cuerpo de la respuesta del servidor como ficheros sobre los que trabajar.

5.5.5. Obtención de precios

La *URI* (`http://server_ip//get_price/package_id`) para obtener el precio de un paquete específico debe incluir el identificador de éste. En base al identificador incluido en la *URI*, se recogen los datos referentes al paquete de la base de datos. Dado que los precios se recogen a través de *Web-Scrapers* también sin una clara relación entre el paquete en *Steam* y el equivalente en la tienda a analizar, estos *pricers* obtienen toda la información conocida por parte de *Steam* para aumentar en la medida de lo posible la precisión con la que se realizan las búsquedas en tiendas externas a *Steam*.

El servidor hace uso de una estructura especial que contiene un `Vector` de estructuras que tengan implementada la interfaz *pricer*. Se itera de forma concurrente sobre este vector de *Pricers* de forma que se consulten simultáneamente tantos como permita la máquina anfitrión, enviando a todos estos el paquete sobre el que obtener la información. El método `get_price_info` devolverá una estructura de datos característica de *Rust* (`Result`) que permitirá filtrar los resultados se tal forma que la lista de precios solo contendrá resultados de tiendas válidos y no valores nulos o inicializados con parámetros no validos que habría que tener en cuenta más adelante.

Esta estructura se envía serializada como un *JSON* al cliente de la petición. Se puede observar un ejemplo de la respuesta del servidor en la figura 5.9.

5.5.6. Obtención de métricas desde HowLongToBeat

El servidor también recopila la duración de un videojuego desde la base de datos de *HowLongToBeat*. *HowLongToBeat* es un sitio web donde los usuarios apuntan el tiempo que les ha llevado finalizar un juego en distintas modalidades (solo la historia principal, recoger los extras...) y realiza una media para que el usuario pueda hacerse una idea del tiempo que puede requerir terminar un juego. En muchas ocasiones los potenciales compradores se encuentran con la duda entre varios títulos y el tiempo de ocio que ofrece cada uno puede ser el valor añadido final en base al que deciden la compra a realizar. Es por esto que se

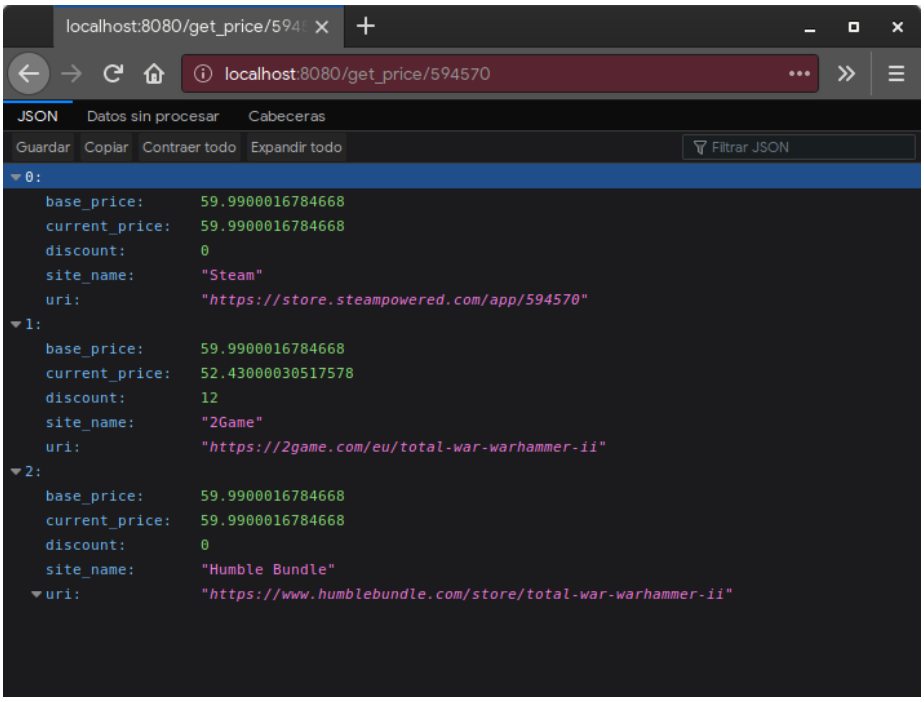

Figura 5.9: *Respuesta del servidor con precios en formato JSON*

ha implementado una biblioteca que es capaz de realizar *web-scraping* en *HowLongToBeat* y devuelve la información en formato *JSON*. Al igual que con las tiendas, se realiza *web-scraping* directamente sobre el sitio web con la información del paquete que se encuentra disponible en la base de datos y en base a eso se obtiene un posible resultado.

Capítulo 6

Aplicación móvil

Para el desarrollo de la aplicación se ha optado por el marco de trabajo *Flutter*, creado por *Google*. *Flutter* lanzó su primera versión estable en diciembre de 2018 y ofrece una solución similar a *React Native* de *Facebook*. Sin embargo utiliza *Dart* como lenguaje de programación en vez de *JavaScript*, y por familiaridad con lenguajes fuertemente tipados y por querer experimentar con este nuevo marco de desarrollo, se ha escogido como el principal para el desarrollo de las aplicaciones móviles.

Flutter permite desarrollar desde cualquiera de las tres plataformas principales y genera código que se ejecuta nativamente en *Android*.

6.1. Selección del nombre

Se ha escogido el nombre de Ofertus porque hace una clara referencia a la palabra oferta, lo cual hace que sea fácil de recordar pero que tenga cierta personalidad.

6.2. Dependencias utilizadas

Para la lógica fuera del contexto de la propia aplicación se han utilizado diversas bibliotecas disponibles para *Flutter*.

6.2.1. cached_network_image

cached_network_image [38] se distribuye bajo licencia *MIT*. Permite la obtención de imágenes abstrayendo la capa de caché local para acelerar el tiempo de respuesta de la interfaz de usuario en caso de que la imagen requerida se encuentre ya almacenada en el dispositivo.

6.2.2. nima

nima [39] se distribuye bajo licencia *MIT*. Anunciada durante el lanzamiento de la versión 1.0 de *Flutter*, *nima* es una biblioteca para la reproducción de *sprites* 2D animados. Sin embargo solo permite reproducir animaciones creadas bajo la misma plataforma web de *Nima*.

6.2.3. sqflite

sqflite [40] se distribuye bajo licencia *MIT*. Permite la ejecución de código *SQL* contra una base de datos local *SQLite*. La biblioteca permite realizar transacciones directas sin ningún tipo de abastracción como *ORM*.

6.2.4. share

share [41] se distribuye bajo licencia *BSD*. Permite conectar elementos de la aplicación a la ventana de compartición de elementos de *Android*. Dependiendo del tipo de elemento que se pretenda compartir *Android* ofrecerá un listado de las aplicaciones que permite compartir ese tipo de elemento.

6.2.5. shared_preferences

shared_preferences [42] se distribuye bajo licencia *BSD*. Permite guardar ajustes simples de aplicaciones de forma local a través del método nativo de *Android* de guardar ajustes. Se utiliza para que el usuario pueda borrar los ajustes y reiniciarlos fácilmente de la forma esperada.

6.2.6. url_launcher

url_launcher [43] se distribuye bajo licencia *BSD*. Biblioteca que lanza la *URL* solicitada a través del navegador web instalado por defecto en el dispositivo.

6.2.7. intl

intl [44] se distribuye bajo licencia *BSD*. Biblioteca que lanza los comandos que se muestran más adelante en la sección 6.8 para la traducción de los ficheros de recursos.

6.3. Diseño de la aplicación móvil

Durante el desarrollo de la aplicación se han realizado varias iteraciones respecto al diseño de esta, tratando de obtener una interfaz lo más simple posible pero lo suficientemente intuitiva para que los usuarios nuevos puedan utilizarla desde el primer momento.

6.3.1. Diseño de la aplicación general

A la hora de diseñar la aplicación, se ha optado por diseños basados en el lenguaje de diseño *Material* de *Google*. Se han utilizado elementos estándares de *Android* debido a la carencia de diseñadores dedicados así como a la falta de experiencia en diseño por parte del único programador de la plataforma.

El diseño de la aplicación ha pasado por varias iteraciones. Los bocetos iniciales trataban de un concepto basado en el clásico menú lateral deslizable con varias secciones como el que se muestra en la figura 6.1(izquierda)

Tras trabajar durante unos días en la aplicación y terminar de organizar las secciones con las distintas partes funcionales de la aplicación, era obvio que un menú lateral ocupaba demasiado espacio para tan pocos ajustes, por lo que se optó por una vista por pestañas como la que se muestra en la figura 6.1(derecha).

Finalmente se optó por un diseño mucho más simple, con una vista principal en columna mostrando los títulos con un tamaño que permita fácilmente distinguirlos y con un acceso

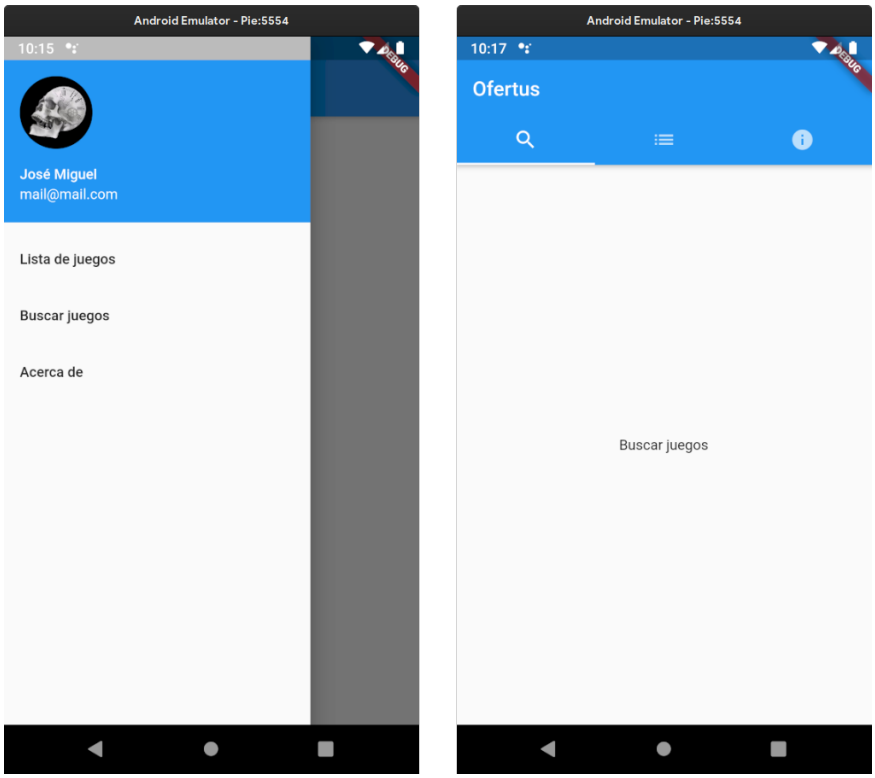

Figura 6.1: *Diseños originales de la aplicación móvil*

a las búsquedas para añadir juegos nuevos como un botón flotante en la esquina inferior derecha. Se añade un pequeño botón en la esquina superior derecha para acceder al resto de apartados de la aplicación: los ajustes y la información de la aplicación. La versión final tiene el aspecto mostrado en la figura 6.2 [1] y como se puede observar, se ha trabajado en ofrecer una interfaz adaptable a temas oscuros de cara al lanzamiento de *Android Q*.

6.3.2. Diseño de los elementos contenedores de paquetes

Se ha trabajado e iterado varias veces sobre la interfaz de usuario con el fin de ofrecer una experiencia de usuario que resulte agradable a la vista, práctica e intuitiva y permita al usuario moverse rápidamente entre los distintos elementos que componen los paquetes para que puedan ver la información requerida rápidamente.

[1] Asobo Studio 2019, A Plague Tale: Innocence, Focus Home Interactive; CREATIVE ASSEMBLY 2019, Total War: THREE KINGDOMS, SEGA; Thunder Lotus Games 2017, Sundered®: Eldritch Edition, Thunder Lotus Games

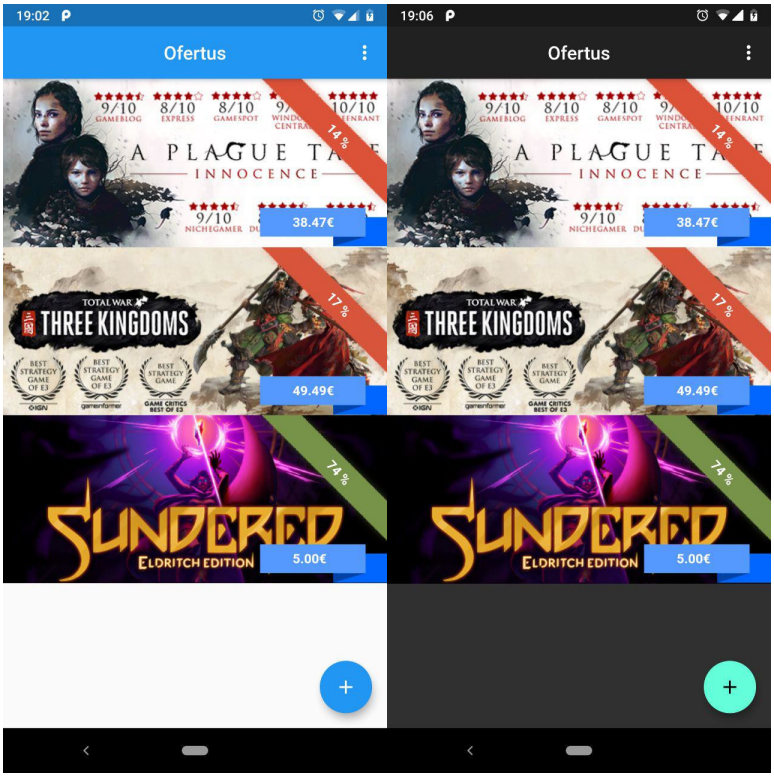

Figura 6.2: *Diseño final de la aplicación*

En su última versión estable, la aplicación móvil cuenta con una vista en forma de lista vertical donde cada elemento contiene un paquete agregado por el usuario para ser supervisado por la aplicación.

Cada elemento visual representativo de cada paquete al que se nombrará a partir de ahora como "*Tile*" contiene tres elementos visuales que permiten reconocer rápidamente el paquete al que hace referencia y la información más relevante respecto al precio para poder así tener una vista global con la situación actual de cada paquete al que el usuario ha decidido realizar un seguimiento desde la aplicación.

Con la intención de desarrollar una aplicación con un diseño modular para que sea más sencilla su modificación o de cara a su mantenimiento, cada `GameTileWidget` se implementa como un *Widget* que se inserta dentro de otro que solo contiene la lógica de interacción (hacer clic para ir a detalles o arrastrar hacia los laterales para eliminar) al que se ha decidido nombrar `InteractivePackageTileWidget`.

`InteractivePackageTileWidget` contiene la lógica de navegación de los elementos que contienen la información de los paquetes, haciendo así que cuando un usuario haga clic en el *Widget*, la aplicación redirija la vista principal de la aplicación a la ventana de detalles del paquete. Esto se realiza mediante la acción `push` del navegador del contexto de *Flutter*, que genera una pila de vistas por las que pasa el usuario para luego poder recorrerla en orden inverso cuando el usuario utilice el botón de ir hacia atrás en su *smartphone*.

Los `GameTileWidget` contiene todo lo referente al aspecto visual de cada *Tile*. Estos se forman por un *Hero component*, un componente proporcionado por el propio marco de desarrollo de *Flutter* que permite animar el componente durante la transición a otro estado. En este caso se utiliza para que el componente visual del paquete se traslade a la cabecera de la vista de detalles de precios del paquete haciendo así una transición más suave entre ambas vistas con el fin de ofrecer una mejor experiencia de usuario en la navegación entre vistas de la aplicación. Los `GameTileWidget` contienen actualmente tres componentes informativos en el apartado visual: una imagen de fondo, el precio actual más bajo y el porcentaje de descuento respecto al respectivo precio original del paquete que se muestra. A continuación se muestra un ejemplo de cómo se renderiza esta información en la figura 6.3

Figura 6.3: *Diseño final de GameTileWidget*

Para la obtención de imágenes de cada paquete se ha decidido utilizar los recursos ofrecidos por la propia web de *Steam*. Se ha decidido utilizar esta base por contener la mayor cantidad de imágenes para representar los distintos paquetes disponibles que se ofrecen a

través de la aplicación. Sin embargo se debe remarcar que *Steam* no ofrece estas imágenes de forma consistente para todos sus paquetes, lo que dificulta la obtención de estos.

Steam divide sus tipos de paquetes en tres tipos distintos: *Apps*, *Subs* y *Bundles*. Y obtiene la imagen en base a una *URI* respectiva para cada tipo de paquete:

```
"Apps" : "https://steamcdn-a.akamaihd.net/steam/apps/
{ID del paquete}/header.jpg"
"Subs" : "https://steamcdn-a.akamaihd.net/steam/subs/
{ID del paquete}/header.jpg"
"Bundles" : "https://steamcdn-a.akamaihd.net/steam/bundles/
{ID del bundle}/{UUID del bundle}/header.jpg"
```

Para el caso de *Apps* y *Subs*, formar una *URI* en base al identificador del paquete y el tipo de paquete que es, se vuelve una tarea trivial, sin embargo para el caso de los *Bundles* se ha requerido modificar la estructura de los paquetes en toda la infraestructura para poder añadir el *UUID* necesario para poder obtener una imagen representativa del *Bundle*. Este *UUID* necesario para poder formar la *URI* se incluye dentro de cada paquete, siendo un texto vacio en el caso de *Apps* y *Bundles* y se recopila en la tarea de *web-Scraping* mencionada en la sección 5.3.

Por otro lado, los *Subs* y los *Bundles* mantienen una proporción de aspecto de imagen distinta a la de las *Apps*. Esto supone un problema a la hora de visualizar de forma consistente las imágenes de los paquetes desde la aplicación móvil, y finalmente se ha optado por rellenar verticalmente el espacio con la imagen, obteniendo un resultado con los extremos laterales recortados tal y como se puede observar en al figura 6.4[2]. A pesar del recorte horizontal del contenido, se puede reconocer rápidamente a qué paquete hace referencia la imagen.

Mostrando el mejor precio y su respectivo descuento

El servidor ofrece una *API REST* a la que poder consultar la información de los precios de cada paquete. Enviando en la petición el identificador del paquete de *Steam*, el servidor devuelve un listado de distintas tiendas con el precio actual, el precio usual y el descuento

[2]CAPCOM Co., Ltd. 2017, RESIDENT EVIL 7 biohazard / BIOHAZARD 7 resident evil, CAPCOM Co., Ltd.

Figura 6.4: *Diseño final de GameTileWidget para Bundles*

actual que tiene el paquete. En base a esta información se utiliza la tienda con el menor precio en ese momento para mostrar los respectivos datos en la página principal donde se muestran todos los paquetes bajo seguimiento en la aplicación. El precio se muestra sobre la imagen de una cinta en la esquina inferior derecha.

Respecto al descuento, este se muestra sobre una cinta en la esquina superior derecha. Sin embargo el texto que contiene el descuento se rota para asemejarse a la rotación que tiene la cinta sobre la que se representa. Además de esto, se realiza una interpolación lineal de colores entre rojo y verde dependiendo del descuento actual del paquete, siendo respectivamente rojo para un descuento nulo y verde para un producto con un descuento completo (el juego es gratuito), permitiendo así de un vistazo rápido conocer qué títulos bajo seguimiento se encuentran con el mayor descuento (y por lo tanto, con la mejor oferta) en ese mismo instante.

Cuando un usuario añade un paquete a la lista de favoritos dentro de la aplicación móvil, tiene una referencia instantánea del precio y del descuento al momento. Simplemente con encontrar la imagen representativa del videojuego con un lazo más verde, podrá saber rápidamente en que situación respecto al precio se encuentra cada paquete.

6.3.3. Información desde ProtonDB

ProtonDB [45] es un sitio que recolecta informes de los usuarios respecto a la compatibilidad de los juegos con la herramienta *Proton* [46] de Steam. *Proton* es una capa de compatiblidad creada por *Valve* que permite la ejecución de juegos creados para *Windows* en distribuciones *GNU/Linux*. Es un proyecto que se encuentra en desarrollo y la compatibilidad con los distintos videojuegos es distinta entre el catálogo de *Steam*. Es por ello que se formó una comunidad que recopila los resultados obtenidos, generando así una media y categorizando los distintos títulos según su compatibilidad. Dado que *ProtonDB* provee su propia *API REST* a la que se puede acceder a la aplicación, este es el único recurso que se solicita a través de red a otro servidor que no sea el propio de Ofertus. Las categorías de *ProtonDB* son *Borked, Bronze, Silver, Gold* y *Platium* siendo *Borked* compatibilidad nula y *Platinum* un rendimiento a la par que en *Windows*. Esta información se muestra dentro del *Widget* de plataformas en la vista de detalle de los paquetes tal y como se muestra en la figura 6.5.

Caché de elementos para la interfaz

Flutter contiene un tipo especial de *Widget* llamado `FutureBuilder` que permite generar elementos visuales que se obtiene de forma asíncrona. Dado que toda la información que muestra la aplicación proviene de Internet y que el renderizado de la aplicación móvil es síncrono, `FutureBuilder` fuerza al programador a dibujar algo mientras se recupera la información final que se quiere mostrar. El problema es que durante el primer *frame* del dibujado, la información no se puede obtener y esto hace que se dibuje un *Widget* vacío. A simple vista no parece un gran problema, pero durante las animaciones y las transiciones como las proporcionadas por el *Hero Widget*, se llama al método de dibujado varias veces, y por lo tanto se provoca un efecto de parpadeo. Para solucionarlo ha sido necesario implementar un sistema de caché local con la información requerida por los `GameTileWidget` para que estos puedan ser dibujados en el primer *Frame* en el caso de existir, solucionando

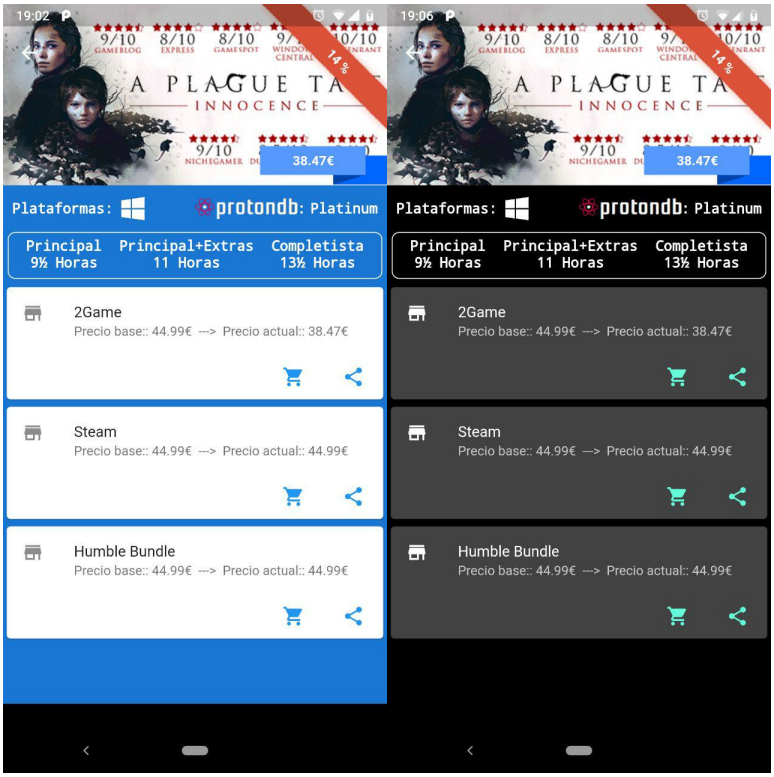

Figura 6.5: *Detalles del paquete con tema claro y oscuro*

así el problema de parpadeo.

6.3.4. Temas adaptables y soporte para Android Q

Se han revisado los distintos elementos de la interfaz visual para verificar que la aplicación es compatible con los temas adaptables que *Google* incluirá con el futuro lanzamiento de *Android Q*.

Como se puede observar en las figuras 6.2 y 6.5, las vistas se adaptan dinámicamente al tema que solicite *Android Q*, y se ofrecen en las variantes claro y oscuro. Hay ciertos elementos donde se han aplicado colores específicamente para evitar que en alguna de las variantes no sean lo suficientemente visibles.

Sin embargo hay un elemento que no ha sido posible solucionar, y este es la cuadro de texto en la vista de búsquedas de la aplicación. Este cuadro de búsquedas no adapta el tema automáticamente y se ha tenido que especificar manualmente para que lo haga, sin embargo,

en su version `1.5.4` aún no es posible alterar manualmente algunos colores de la cabecera, y se esta haciendo un seguimiento de esta funcionalidad en la página de errores de *Flutter* en *GitHub* [47]

6.4. Búsquedas

Flutter ofrece el acceso a una vista de búsquedas básica inspirada por el lenguaje de diseño *Material Design* de *Google*, sin embargo no ofrece ningún tipo de implementación de motor de búsquedas para trabajar por lo que ha sido necesario implementar uno propio. A la hora de realizar las búsquedas se optó por un sistema similar al que realizan los motores de búsqueda modernos en la web, ofreciendo resultados según el usuario escribe la consulta en vez de requerir que éste escriba la consulta y entonces decida realizar la búsqueda, para ofrecer así una más rápida búsqueda de elementos y una mejor experiencia de usuario.

Dado que las búsquedas se realizan de forma local en el dispositivo móvil y que el número de entradas sobre los que hay que realizar la búsqueda es superior a las 40.000 se requiere de un motor de búsquedas rápido pero fiable. Inspirado en las prácticas sobre búsquedas de documentos realizadas en la asignatura de "Sistemas de gestión de datos y de la información", se implementó un motor de búsquedas basado en índices invertidos y se añadió un componente de nodos *Trie* para acelerar las búsquedas.

Para comprender el motor de búsquedas, es necesario entender los ficheros necesarios y cómo se cargan estos. El motor de búsquedas requiere de dos listas de información que el servidor distribuye como dos ficheros en formato *JSON*. Estos ficheros se denominan como `gameDB.json` y `invertedIndex.json`.

6.4.1. gameDB.json

El fichero `gameDB.json` incluye un diccionario donde la clave es el identificador del paquete según *Steam* y el valor un objeto que contiene toda la información con la que se trabaja en la aplicación. Este fichero se utiliza como principal base de datos dentro de la

aplicación móvil. Cada paquete contiene una estructura como la que se puede ver en el listado 6.1.

Listado 6.1: *Ejemplo de paquete en gameDB.json*

```
"986830":
    {"title":"The Legacy: The Tree of Might",
    "package_type":"app",
    "bundle_id":"",
    "is_win":true,
    "is_mac":true,
    "is_linux":false
    }
```

6.4.2. invertedIndex.json

El fichero `invertedIndex.json` contiene un diccionario donde para cada palabra contenida entre los distintos títulos de los paquetes, se asocian los identificadores de los juegos que incluyen esta en su título. La estructura de cada elemento tiene el formato que se puede ver en el listado 6.2.

Listado 6.2: *Ejemplo de entrada en invertedIndex.json*

```
"warped":[604030,266730],
"opposing":[50,9340],
"xmas":[993241,913611,765440,560050,
775390,569740,765420,993240,769390],
"shenzhen":[504210],
"lyne":[266010]
```

6.4.3. Carga de los ficheros en memoria

Durante el arranque de la aplicación en frío, se realizan una serie de comprobaciones y actualizaciones en caso de ser necesario respecto a los ficheros correspondientes a las bases de datos y acto seguido, estos se cargan en memoria.

Durante el primer arranque, la aplicación solicitará los ficheros necesarios para el funcionamiento de la aplicación, pero tras esta carga inicial cuando la aplicación de utiliza por primera vez, se realiza una serie de pasos extra.

Con la intención de reducir al máximo las llamadas al servidor, la aplicación cuenta con una cantidad de tiempo mínima que debe pasar antes de volver a preguntar al servidor si

existen actualizaciones en la base de datos. En el caso de que este tiempo se haya sobrepasado, se pregunta al servidor por el *timestamp* de los ficheros de bases de datos para contrastarlo con la información local y poder saber así si la base de datos remota ha sido actualizada o no. Si no ha sido actualizada, no hay necesidad de descargas la base de datos en la aplicación, en caso contrario se descargaran los ficheros en el almacenamiento local y finalmente se cargarán en memoria mientras se muestra una *Splash Screen*. Una vez todos los ficheros necesarios han sido cargados en memoria, se puede utilizar la aplicación.

La carga de ficheros en memoria ocurre en varios pasos. Primero se decodifican los ficheros *JSON* descargados y se guardan como `HashMaps` en memoria. Finalmente se genera un *Trie Map* utilizando la implementación propia que se ha creados para este proyecto, insertando dentro de esta todas las claves del `HashMap` del índice invertido.

6.4.4. Motor de búsquedas

El motor de búsquedas se ha implementado de forma asíncrona para evitar el bloqueo de la interfaz de usuario mientras se recuperan los resultados de la base de datos. Sin embargo, en las pruebas realizadas en los dispositivos disponibles así como en el emulador de *Android*, el motor se ha optimizado lo suficiente para poder devolver resultados sin necesidad de tiempos de carga.

La *API* de *Flutter* llamará a un método en el motor de búsquedas cada vez que la consulta escrita por el usuario cambie, ya sea por haber añadido o eliminado caracteres en la entrada de datos. Esta entrada se separa en las distintas palabras (completas o parcialmente escritas) separando la entrada por espacios y caracteres no alfanuméricos.

Por cada palabra escrita, el motor de búsquedas recorre el árbol de nodos *trie* en base a la palabra y se recopilan todas las palabras que se pueden generar en base a la palabra introducida por el usuario como si de un prefijo se tratara. Por cada cadena de caracteres se genera un listado de palabras que se utiliza con el índice invertido para obtener todos los identificadores de paquetes que contienen en sus títulos dichas palabras. Tras ello se recogen

los paquetes respectivos a los identificadores obtenidos y en el caso de que haya más de una cadena en la entrada del usuario se realiza un `Set` conteniendo la intersección de paquetes obtenidos en las diferentes listas generadas por cada cadena. El uso conjunto de índices invertidos y nodos *trie* se lleva utilizando desde años y se ha demostrado su eficacia tal y como describe [48].

Finalmente se filtra el listado de resultados en base a las preferencias del usuario respecto a los sistemas operativos compatibles para los que éste ha escogido mostrar o no mostrar posibles resultados. Si se da el caso de que una búsqueda no obtiene ningún resultado, si además el usuario haya activado los filtros de búsqueda como la plataforma, se muestra una línea de texto indicando los filtros activos en ese momento para recordar al usuario los posibles motivos por lo que su búsqueda no ha logrado obtener ningún resultado.

6.5. Trabajando de forma asíncrona

La experiencia de usuario se ha colocado como uno de los pilares fundamentales del desarrollo de este proyecto para poder ofrecer una solución equivalente a la de una aplicación profesional similar a las que los usuarios puedan esperar de una gran empresa.

Para evitar que la lógica de la aplicación pueda bloquear la interfaz gráfica de la aplicación y por lo tanto provocar que la aplicación en algún momento no responda a los gestos del usuario, toda interacción entre los hilos de ejecución de la interfaz gráfica y la lógica de la aplicación se ha implementado utilizado mecanismos asíncronos, asegurando así que el usuario siempre reciba algún tipo de retroalimentación a sus interacciones aunque la lógica de la aplicación no haya finalizado de realizar las tareas solicitadas. Realizar estas interacciones de forma aśincrona fue una prioridad desde el primer momento por el alto uso de conexiones a servidores y servicios de terceros que requiere la app donde los tiempos de respuesta pueden llegar a ser demasiado altos.

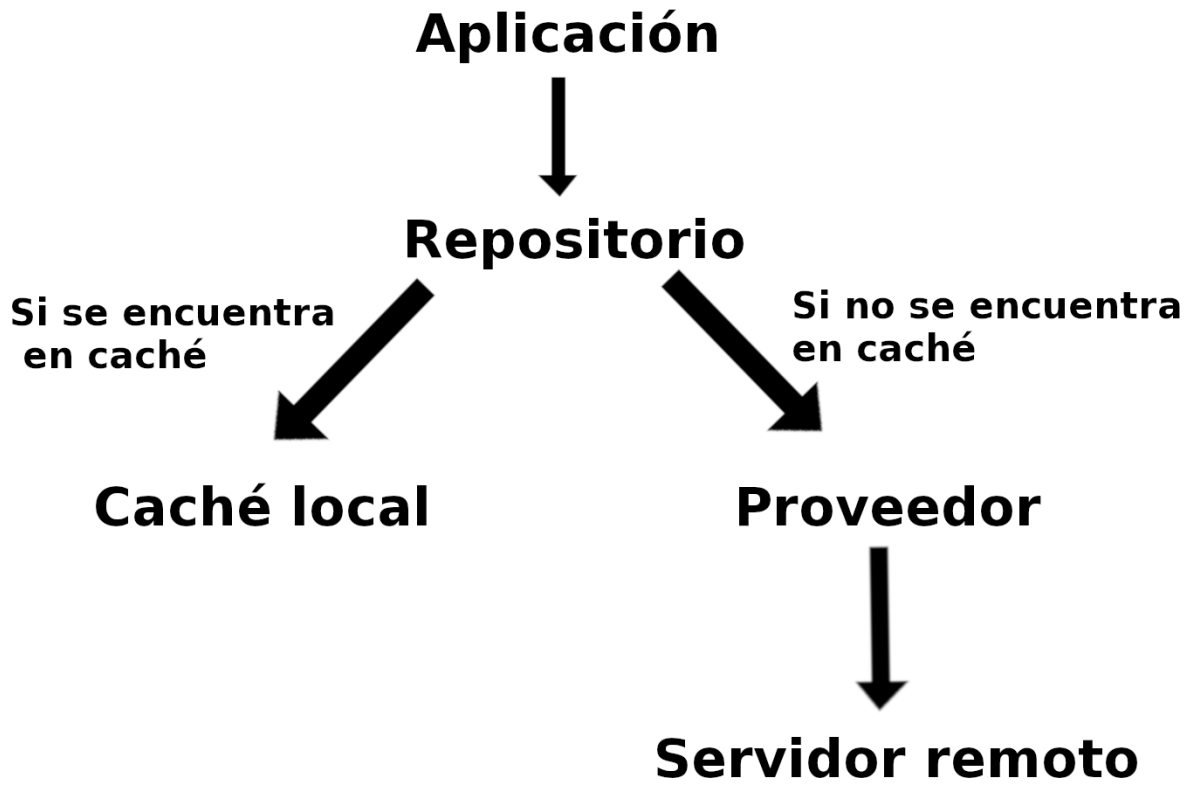

Figura 6.6: *Diseño del repositorio de datos*

6.6. Diseño del acceso a los datos

Para reducir la carga en el servidor, mientras la aplicación se mantiene en memoria, esta contiene una caché local donde almacena los precios y la distinta información de los paquetes para que estén disponibles instantáneamente en futuras peticiones. Esta caché se regenera cada vez que la aplicación se inicia y el acceso a esta queda abstraído por el uso de repositorios, obteniendo la información tal y como se muestra en la figura 6.6.

Los distintos elementos de la aplicación solicitarán los datos de un paquete al correspondiente repositorio y será tarea de este determinar si dicha información ya se encuentra en caché y recuperarla directamente desde ahí, o en el caso contrario llamar al proveedor de servicios web para obtener la información desde el servidor propio de Ofertus. De esta manera el acceso a la información queda unificado por un mismo patrón y toda la lógica de conexión externa queda aislada de la lógica de la aplicación.

6.6.1. Creación y uso de la caché

La caché es una estructura que contiene un `HashMap` donde la clave es el identificador del paquete, y el valor los datos de este. La aplicación contiene actualmente un sistema de caché para la información de los precios y para la información de la duración del título ofrecida por *HowLongToBeat*, reduciendo así tanta carga como sea posible en el servidor. Si el usuario de la aplicación desea refrescar los datos deberá eliminar la entrada del paquete de su listado de favoritos y volver a añadirla y simplemente reiniciar la aplicación.

6.7. Estado del soporte multiplataforma

Aunque *Flutter* ofrece un marco de desarrollo multiplataforma para una única base de código compartida entre ambas plataformas, la realidad es distinta en el momento en el que el proyecto requiere de configuraciones o extensiones avanzadas. Sin acceso a un dispositivo basado en el sistema operativo *iOS* de *Apple* y sin una cuenta de desarrollador debido al alto coste que ello implica, no se ha podido comprobar el estado de la compatibilidad de la aplicación en dispositivos de *Apple*. Sin embargo se ha podido comprobar que la aplicación, en su estado actual funciona sin ningún problema en un emulador de *iOS* como se puede comprobar en la figura 6.7[3].

Por otro lado se encuentra la situación de la interfaz gráfica, donde se han utilizado elementos basados en *widgets* que siguen el lenguaje de diseño *Material Design* de *Google*. Aunque estos *widgets* son compatibles con dispositivos *Apple*, una adaptación a los *widgets* basados en el lenguaje de diseños de *Apple*, conocidos como *Cupertino*, sería necesario para que la aplicación tenga un aspecto nativo en este tipo de dispositivos.

[3]Nomada Studio 2018, GRIS, Devolver Digital; Paradox Development Studio 2019, Imperator: Rome, Paradox Interactive

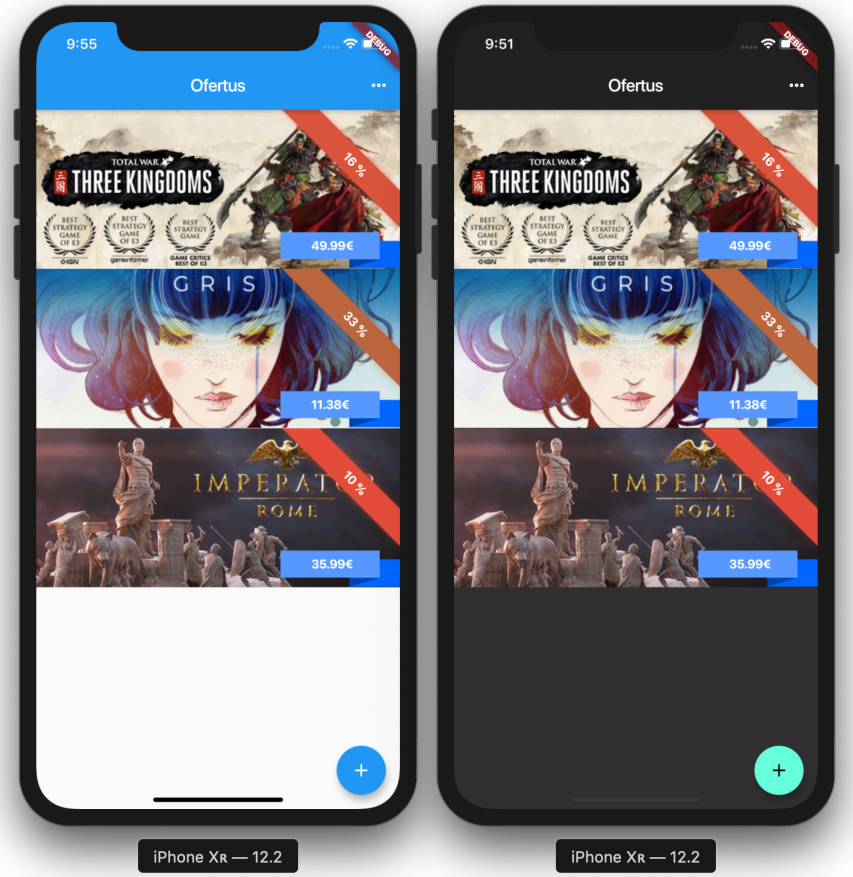

Figura 6.7: *Aplicación funcionando en emulador de dispositivo iOS*

6.8. Internacionalización

La aplicación ha sido preparada para el uso de múltiple idiomas para su interfaz gráfica. La internacionalización se ha implementado para que la comunidad pueda trabajar de forma voluntaria en la traducción a múltiples idiomas. Originalmente se implementó un sistema basado en ficheros *JSON* por su simplicidad, pero finalmente se optó por el sistema oficial recomendado por *Flutter* basado en ficheros *ARB* (*ApplicationResourceBundle*) para tratar de seguir la especificaciones recomendadas tanto como sea posible.

Como primer paso, se han utilizado tantos *Widgets* de *Flutter* y su biblioteca *Material* como ha sido posible. Al añadir la dependencia de `flutter_localizations` desde el propio

SDK, se incluyen traducciones para algunos elementos como por ejemplo, el texto por defecto que aparece en el cuadro de inserción de texto para las búsquedas. Sin embargo para la mayoría de elementos de la interfaz gráfica, se ha requerido de un sistema de traducciones propio.

Este sistema esta principalmente organizado dentro de la clase `OfertusLocalizations` y contiene métodos *getter* para cada texto que requiera de versiones traducidas a múltiples idiomas. La principal diferencia entre el diseño basado en ficheros *JSON* y *ARB* es que el primero requiere de menos trabajo pero recoge los recursos de idioma en tiempo de ejecución, mientras que la segunda alternativa implica una mayor carga de trabajo, pero permite obtener los elementos como variables de la clase, asegurando el funcionamiento en tiempo de compilación.

Esta clase se carga dentro de los `localizationsDelegates` de la aplicación junto a los delegados de localizaciones globales y de los *widgets Material*. Esto permite acceder a los elementos de internacionalización desde los *widgets* y las vistas de la aplicación.

Además de esto se deben añadir los elementos `Locale` dentro del objeto `MaterialApp` en la sección de `supportedLocales` de la aplicación, para que la aplicación pueda saber qué idiomas están soportados, puesto que el idioma de la aplicación se selecciona automáticamente en base al utilizando en el sistema operativo, y no es un ajuste propio dentro de la propia aplicación móvil.

El procedimiento para realizar una nueva traducción o añadir nuevos elementos a esta es la siguiente:

1. Añadir el texto como un *getter* dentro de la clase de `OfertusLocalizations`, añadiendo una versión de *fallback*, que en este caso será en inglés.

2. Utilizar la herramienta `intl_translation` que se incluye como una dependencia de desarrollo en la aplicación para extraer los recursos de texto a un fichero *ARB*.

```
flutter pub run intl_translation:extract_to_arb \
--output-dir=lib/l10n \
lib/src/resources/localizations.dart
```

3. Copiar el fichero `intl_messages.arb` por cada idioma que se quiera incluir y renombrar el fichero respecto al idioma que se vaya a implementar (En este caso `intl_en.arb` y `intl_es.arb`).

4. Traducir los ficheros a su respectivos idiomas. Puede realizarse manualmente con un editor de texto o puede utilizarse el *Google Translation Kit* [49] que es compatible con el tipo de ficheros *ARB*.

5. Volver a utilizar la herramienta `intl_translation` para generar los ficheros de código en base a los *ARB* con el comando:

```
    flutter pub run intl_translation:generate_from_arb \
--output-dir=lib/l10n --no-use-deferred-loading \
lib/src/resources/localizations.dart lib/l10n/intl_*.arb
```

6. Utilizar los recursos de la clase en la aplicación móvil, por ejemplo con la variable: `OfertusLocalizations.of(context).searchBestDealFor`

Al repetir estos pasos más adelante, los textos ya generados se conservan, por lo que solo hay que alterar el nuevo contenido añadido o los nuevos idiomas que se quieran añadir.

Capítulo 7

Uso y desplegado del sistema

7.1. Uso y desplegado del servidor

Antes de nada, recordar que todo el código fuente y ficheros relacionados con el proyecto se encuentran disponibles en `https://gitlab.com/ofertus` y todas las instrucciones están pensadas respecto a una copia local del contenido que se encuentra subido en el repositorio.

Para que un usuario pueda poner en marcha el sistema, el primer paso será la preparación del servidor con su correspondiente base de datos. Para la base de datos se requerirá de un sistema *PostgreSQL*, puede ser una instalación nativa dentro del sistema anfitrión, un contenedor a través de algún servicio de virtualización como *Docker* o en una maquina remota, el sistema solo requiere acceso a través del puerto de escucha público que tiene el servidor.

En esta sección se explicará la puesta en marcha como un contenedor de *Docker* puesto que se trata del sistema más agnóstico a la plataforma y las instrucciones deberían ser la misma sin importar dónde se esté ejecutando el proyecto.

7.1.1. Instalación de Docker

Desde la web de *Docker*, el usuario debería seguir las instrucciones de instalación disponibles para su sistema operativo. En el caso de que utilice una distribución de *Linux* no soportada oficialmente (como es el caso de *Arch Linux*), es probable que los propios medios

de distribución de paquetes de dicha distribución contenga un paquete de *Docker* a instalar.

Remarcar que, en el caso de *Linux*, para poder trabajar con *Docker* sin requerir permisos de superusuario para cada operación, el usuario deberá añadirse dentro del grupo `docker` que se habrá creado en su equipo. Esto se puede realizar con el siguiente comando (requiere de privilegios de superusuario):

```
gpasswd -a $USER docker
```

El usuario deberá reiniciar el equipo para que se apliquen los cambios. En muchos casos es posible que el servicio de *Docker* que se debe ejecutar en segundo plano no lo haga automáticamente tras la instalación. En el caso de distribuciones *Linux* basadas en *SystemD*, el proceso de arranque se puede auto-iniciar con el comando (requiere de privilegios de superusuario):

```
systemctl enable docker
```

Si el usuario lo prefiere, puede mantener el auto-arranque del servicio desactivado y activarlo manualmente solo cuando lo necesite con el comando (requiere de privilegios de superusuario):

```
systemctl start docker
```

7.1.2. Creación del contenedor de PostgreSQL

Crear el contenedor para el sistema de base de datos es un proceso sencillo si Docker se ha instalado adecuadamente, el primer paso será descargar la imagen que contiene los datos del contenedor:

```
docker pull postgres
```

Una vez hay finalizado, se debe poner en marcha el contenedor a través del comando:

```
docker run --name $NOMBRE_CONTENEDOR \
-e POSTGRES_USER=$POSTGRE_USER \
-e POSTGRES_PASSWORD=$PASSWORD \
-e POSTGRES_DB=$NOMBRE_DB \
-p 5432:5432 \
-v $DIR_VOLUMEN:/var/lib/postgresql/data \
-d postgres
```

A continuación se explican las distintas variables de este comando y su propósito:

$NOMBRE_CONTENEDOR: Nombre con el que reconocer el contenedor en la máquina local.

$POSTGRE_USER: Nombre de usuario que se utilizará para acceder a la base de datos de *PostgreSQL*.

$PASSWORD: Contraseña de acceso al sistema de base de datos.

$NOMBRE_DB: El nombre de la base de datos que se utilizará dentro del sistema de la base de datos.

5432: El puerto dentro del contenedor (que se exhibirá a través del mismo puerto en la máquina local).

$DIR_VOLUMEN: El directorio local donde el contenedor escribirá los datos del contenedor de forma persistente.

-d: Ejecutar el contenedor en segundo plano.

postgres: El nombre de la imagen sobre la que se construirá el contenedor.

Con esto, el sistema de base de datos ya estaría creado y funcionando, escuchando a través del puerto `5432`. Si el usuario requiere reiniciar o volver a arrancar el contenedor, Docker mantiene la configuración de lanzamiento dentro del contenedor, por lo que tan solo necesitará ejecutar el comando:

```
docker start $NOMBRE_CONTENEDOR
```

7.1.3. Configuración de los proyectos para el uso de la base de datos

De los proyectos de *Rust* creados, aquellos que requieren de conexión contra la base de datos incluyen un fichero `.env` en su correspondiente directorio con la configuración de conexión a la base de datos. Esta deberá actualizarse para reflejar la configuración que el usuario ha decidido utilizar. La configuración por defecto es:

```
DATABASE_URL=postgres://postgres:passwd@localhost:5432/ofertus
```

Donde `postgres:passwd` son `$POSTGRE_USER:$PASSWORD`, `localhost` se trata de la dirección IP donde se haya la base de datos, `5432` el puerto al que debe conectarse y `ofertus` es `$NOMBRE_DB`.

7.1.4. Aplicar los esquemas de la base de datos

Primero se deberá instalar la herramienta de *CLI* de *Diesel* y tal como se explica en el apartado 3.1.3. Con la herramienta de *Diesel* instalada, desde el directorio del proyecto llamado `diesel`, ejecutar el comando:

```
diesel setup
```

7.1.5. Inserción de entradas en la base de datos

Desde el directorio del proyecto `steam_craper`, lanzar el proyecto con:

```
cargo run [--release] -- -t $NUMBER_THREADS
```

Cargo compilará y ejecutará el scraper de *Steam* utilizando el número de hilos definidos por la variable `$NUMBER_THREADS`. Si se desea utilizar tantos hilos como permite el sistema, se puede lanzar el programa sin parámetros directamente con:

```
cargo run [--release]
```

Utilizando el parámetro opcional `release`, el programa se ejecutará con optimizaciones en el código para el tiempo de ejecución pero puede requerir de más tiempo para compilarse.

7.1.6. Arranque del servicio del servidor

El servidor al igual que el *scraper* se puede arrancar enviando como parámetros el número de hilos de ejecución y el puerto de escucha de peticiones *HTTP*. Cada hilo de ejecución procesará independientemente del resto las llamadas *HTTP* realizadas por distintos clientes, por lo que a más hilos, mejor se podrá trabajar contra mayores cantidades de clientes simultáneos.

Desde el directorio del proyecto, lanzar el servidor con:

```
cargo run [--release] -- -p $PORT -t $NUMBER_THREADS
```

Si no se utilizan parámetros, el servidor se ejecutará con el número de hilos de ejecución disponibles en la maquina anfitrión y escuchando en el puerto `8080`. Una vez que el servidor empiece a funcionar, mostrará a través de la salida estándar un *log* con las distintas operaciones solicitadas por los usuarios. Si el usuario desea utilizar el servicio de *Sentry*, deberá configurar la clave de *API* en el código de la aplicación.

7.2. Desplegado del servidor con Docker Compose

La otra vía para poner en marcha un servidor de forma local, es a través de *Docker Compose*. Con todo lo necesario para poner en marcha un servidor pre-configurado y compilado en una imagen *Docker* todo lo que se requiere es crear un fichero `docker-compose.yml` con el siguiente contenido y lanzar el contenedor con `docker-compose up`. *Docker Compose* se encargará de descargar las imágenes necesarias, de conectar los contenedores entre sí y de dejar el proceso del servidor en marcha.

```
    version: "3.7"
services:
  server:
    image: alosarjos/ofertus-docker:v1.7
    environment:
      - DATABASE_URL=$DATABASE_URL
      - SENTRY_DSN=$SENTRY_DSN
      - "80:80"
    links:
      - "db:database"
  db:
    image: postgres:latest
    environment:
      POSTGRESS_PASSWORD: passwd
      POSTGRES_USER: postgres
      POSTGRES_DB: ofertus
volumes:
  data-volume:
```

7.3. Compilación e instalación de la aplicación

Si se desea instalar la aplicación desde el código fuente de la aplicación, los pasos a seguir son los siguientes:

Preparar el entorno de desarrollo y tal como se describe en la sección 3.2. Para la aplicación se requieren el *SDK* de *Android* y *Flutter* nada más. Recordar que se deben ajustar las variables de entorno tal como se describen en su correspondiente sección.

Para compilar la aplicación, habrá que ir al directorio del proyecto y ejecutar

```
flutter build apk
```

La primera ejecución instalará la herramienta de compilación para aplicaciones A*ndroid Gradle* así como las librerías externas utilizadas, por lo que se recomienda realizarlo con una buena conexión a Internet.

Dado que las propias librerías de *Flutter* y *Google* contienen código obsoleto, es posible encontrar mensajes de advertencia durante la compilación.

Para poder instalar la aplicación en el dispositivo móvil, es necesario activar la depuración USB en este.

Activar las opciones de desarrollador en un dispositivo Android

Ir a `Ajustes` → `Sistema` → `Información del teléfono` y pulsar sobre número de compilación múltiples veces hasta que aparezca un mensaje indicando que las opciones de desarrollador han sido activadas.

En la nueva sección de desarrollo dentro de los ajustes del dispositivo, activar la depuración USB tal y como se puede ver en la figura 7.1.

Una vez los ajustes han sido activados, se deberá conectar el dispositivo *Android* al PC y un mensaje en el dispositivo móvil pedirá permisos para autorizar a esa máquina a poder depurar en ese teléfono (ver Fig 7.2). Es necesario que el usuario acepte.

El siguiente paso es asegurarse de que *ADB (Android Debug Bridge)* reconozca el dispositivo. Con este conectado, basta con lanzar el comando

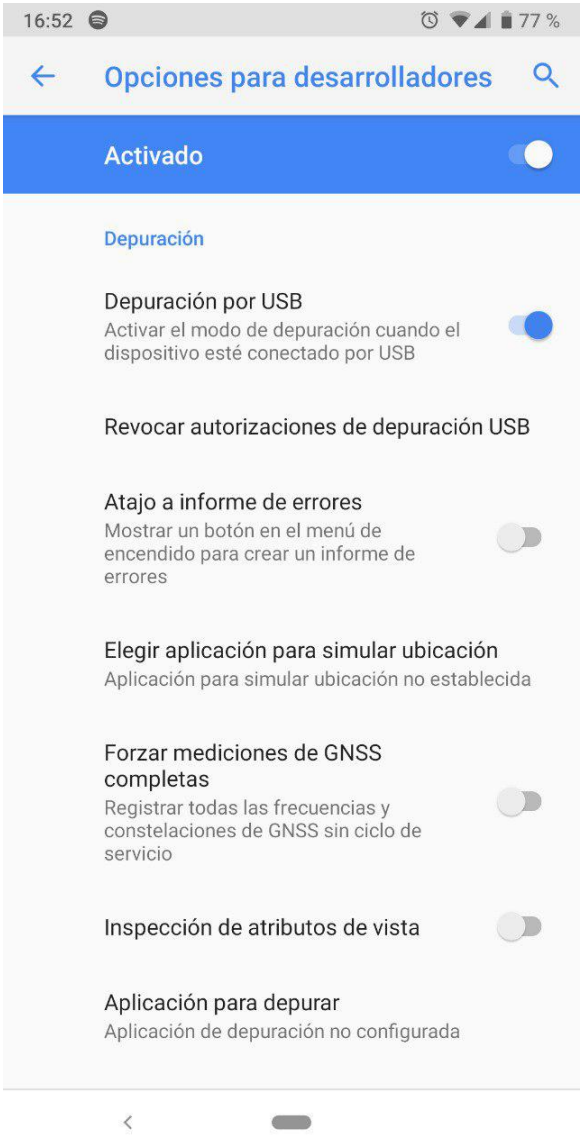

Figura 7.1: *Depuración por USB en los ajustes de Android*

```
adb devices
```

el cual debería mostrar un listado de dispositivos accesibles tal y como se puede observar en la figura 7.3.

Una vez todo está preparado, se puede compilar y lanzar la aplicación en el dispositivo móvil con el comando:

```
flutter run
```

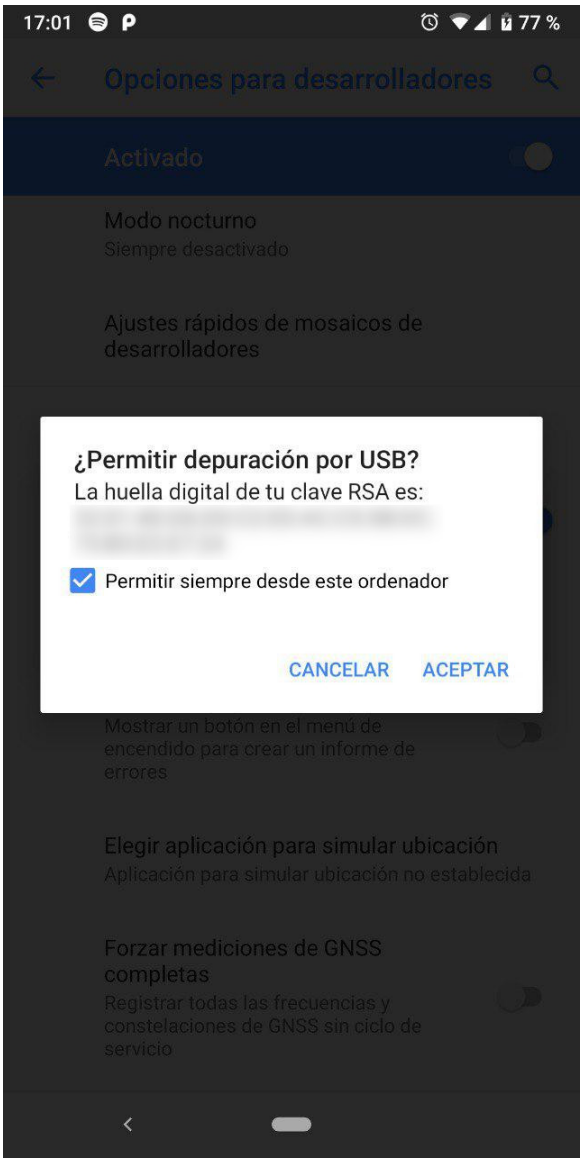

Figura 7.2: *Depuración por USB en los ajustes de Android*

La aplicación se ejecutará en modo de depuración en el dispositivo pero quedará instalada y se podrá utilizar incluso después de desconectar el teléfono de la máquina.

Una vez la aplicación ha sido instalada, el usuario deberá seleccionar el icono de esta para inicializarla y deberá escoger una cuenta de *Google* de entre aquellas que se encuentren en ese momento activas en ese dispositivo móvil.

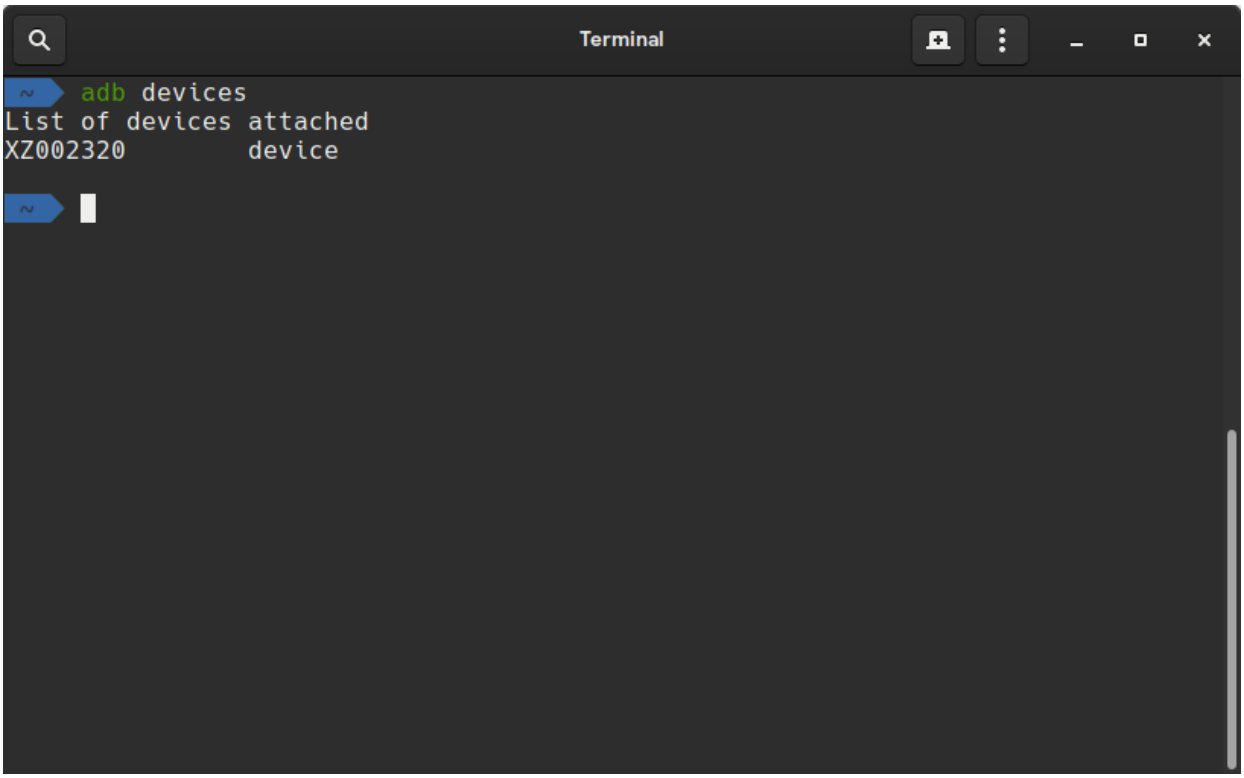

Figura 7.3: *Depuración por USB en los ajustes de Android*

7.4. Uso de la aplicación móvil

7.4.1. Acceso a los ajustes

Una vez iniciada la sesión, podrá acceder a los ajustes haciendo clic en el botón que se encuentra en la esquina superior derecha y entrando en la sección de ajustes. Ahí encontrará los ajustes de filtros por plataforma que le permitirán seleccionar para qué plataformas específicas desea poder encontrar resultados durante la búsqueda de paquetes.

7.4.2. Búsqueda de paquetes

Desde la pantalla principal de la aplicación, podrá acceder al buscador de paquetes desde el botón que se encuentra en la esquina inferior derecha.

Una vez en la sección de búsqueda, solo deberá escribir el nombre del paquete que desee buscar. El motor de búsquedas se ha implementado para que su uso por parte del usuario

sea de la siguiente manera:

El usuario comienza a escribir su consulta y según va escribiendo, aparecerán sugerencias. Los requisitos para una búsqueda exitosa y rápida consisten en escribir correctamente tanto como sea posible del comienzo de cada palabra, esto es:

Si el usuario busca `Half`, la aplicación mostrará todos los resultados que contengan la palabra `half` o cualquier otra que comience por `Half` en su título tal y como se puede ver en la figura 7.4.

Si el usuario esta interesado en buscar un paquete cuyo título sea largo como `Half-Life: Blue Shift`, podrá realizar esta búsqueda rápidamente con la consulta `Half Bl` por ejemplo. Esto permite realizar búsquedas más rápidamente sin necesidad de conocer exactamente el título del paquete tal y como se muestra en la figura 7.5.

7.4.3. Vista principal con paquetes agregados

Una vez seleccionado el paquete desde la lista de sugerencias, la aplicación obtendrá una imagen de cabecera y comenzará el proceso de obtención de la información de los precios de forma asíncrona. Una vez la aplicación tenga los datos de los precios mostrará el mejor precio con el respectivo descuento en la tienda a través de una animación. Una vez agregados varios paquetes, la vista principal queda en forma de columna con cada paquete como se muestra a continuación en la figura 7.6.

Cada uno de estos elementos contiene un acción deslizable hacia los laterales para ser eliminado en caso de haber adquirido ya el paquete o si el usuario deja de estar interesado como se puede ver en la figura 7.7, esta acción puede realizarse en ambos sentidos. Por otro lado también contienen una acción seleccionando el paquete a partir del que se realiza una animación de transición para ofrecer una mejor experiencia de usuario al moverse a través de las distintas vistas de la aplicación.

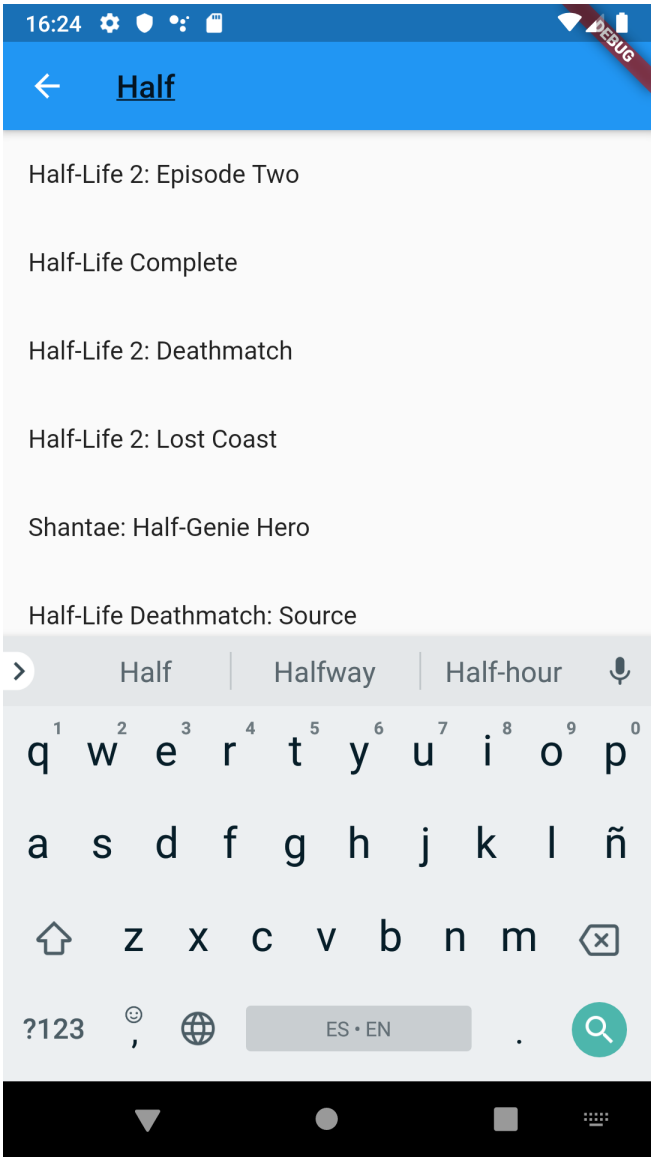

Figura 7.4: *Resultados de una búsqueda parcial en la aplicación*

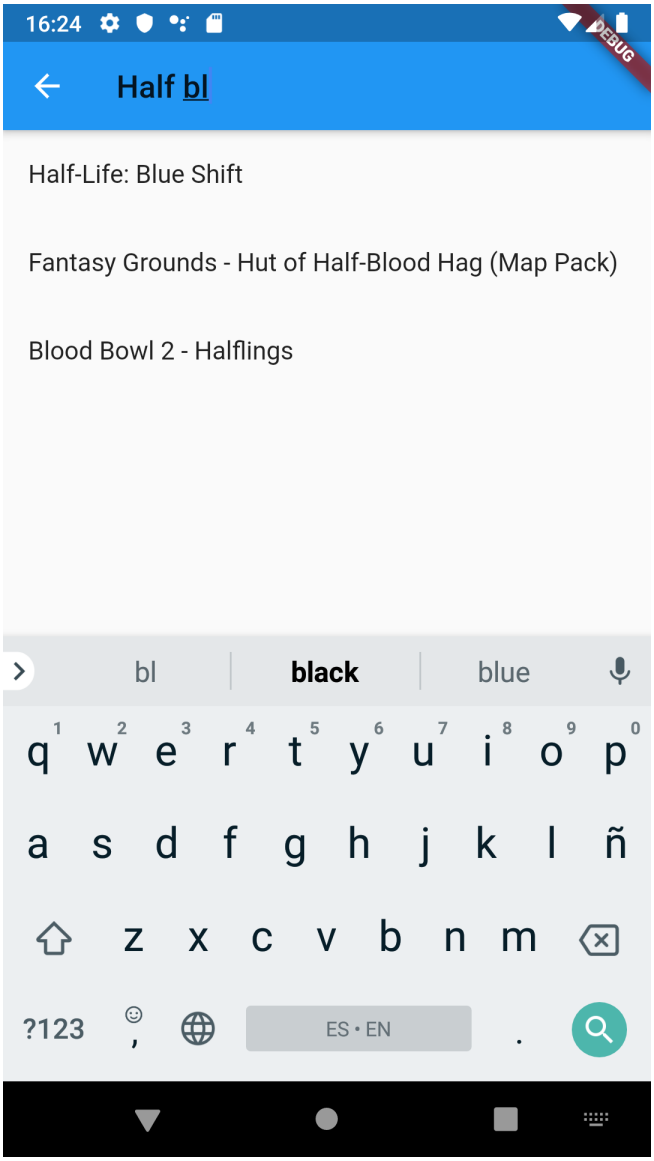

Figura 7.5: *Resultados de una búsqueda parcial con múltiples palabras en la aplicación*

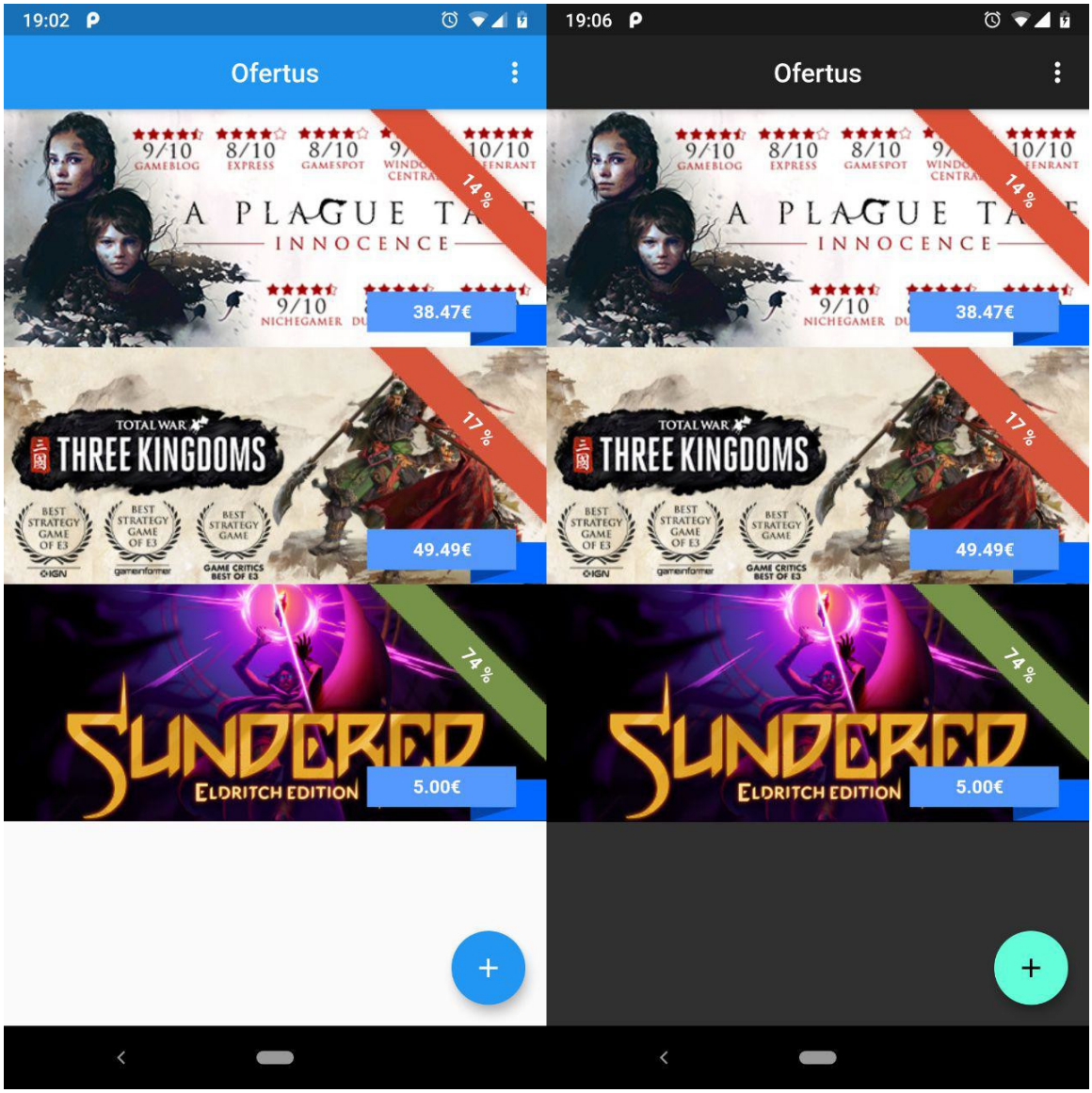

Figura 7.6: *Vista en columna de paquetes añadidos*

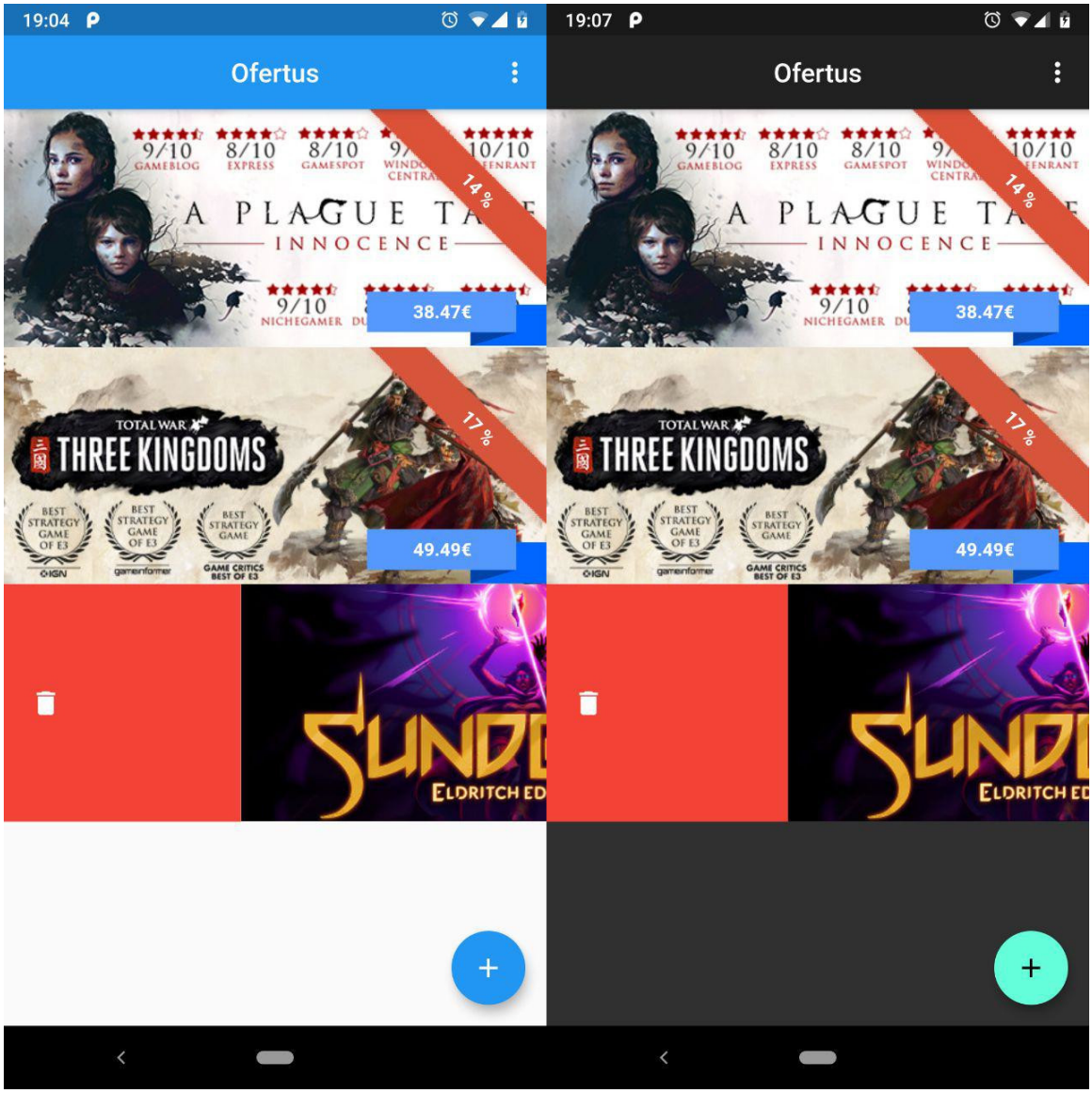

Figura 7.7: *Deslizamiento lateral para eliminar un paquete*

Capítulo 8

Despliegue en la nube y retroalimentación

Cuando se alcanzó el hito de primera *Alpha* se desplegó el servidor dentro de un contenedor de *Docker* en la plataforma de *Google Cloud* y se envió un *APK Android Aplication Package* a un circulo cerrado de usuarios con dispositivos basados en *Android* para recopilar información sobre el estado actual de la aplicación.

8.1. Preparación para el alpha testing

Para preparar una fase de pruebas y comprobar cómo se comporta el software en un contexto de producción, se ha iniciado el proceso de etiquetado de versiones en los distintos productos conocidos. Etiquetando a través de *GIT* y del propio *Rust* cada librería, se logra un mejor control sobre exactamente qué código se esta ejecutando en remoto. Para lograr que el árbol de dependencias funcione adecuadamente se ha utilizado el mecanismo ofrecido por *Cargo*, que permite señalar el origen de una dependencia externa a un repositorio *GIT* especificando la etiqueta del *commit* al que hace referencia la versión requerida. Esto implica una mayor carga de trabajo sobre todo al actualizar bibliotecas utilizadas de forma muy profunda como la librería *ORM* que marca qué puede y qué no puede hacer la estructura `Paquete` para toda la solución, forzando a actualizar las versiones de las bibliotecas que requieren hacer uso de los cambios. Si las bibliotecas contienen versiones diferentes entre sí

de la biblioteca que especifica la estructura `Paquete`, *Rust* lo tomará como una estructura diferente y no compilará el proyecto.

Con las bibliotecas preparadas en una versión lo suficientemente probadas de forma local, es necesario poder realizar un despliegue para que se pueda acceder al servidor desde el exterior. El primer paso fue configurar todo el conjunto del servidor dentro de una imagen de *Docker*. Se ha escogido *Docker* porque es la solución más utilizada para el despliegue de servicios en la nube además del servicio mejor soportado por la mayoría de plataformas y servicios web.

8.1.1. Preparación de la imagen Docker

El primer paso para preparar una imagen *Docker* que contenga la aplicación del servidor es la creación de un `Dockerfile`, un fichero que contiene las instrucciones precisas para la creación de la imagen Docker.

El fichero `Dockerfile` incluye instrucciones en el siguiente procedimiento:

1. Obtener la imagen de Docker que contiene Rust preinstalado
2. Instalar dentro de esta las distintas bibliotecas de desarrollo requeridas para la compilación del servidor y las distintas dependencias.
3. Clonar la versión 0.2.4 del `steam-scraper` y compilar el proyecto en modo `release`
4. Clonar la versión 0.3.6 del servidor y compilar el proyecto en modo `release`
5. Instalar la última versión de la herramienta CLI de Diesel
6. Obtener la imagen base de `Ubuntu 19.04`
7. Instalar las dependencias requeridas
8. Copiar los binarios compilados dentro del contenedor de `Rust` dentro de la imagen de `Ubuntu`

9. Dar permisos de ejecución a los binarios

10. Clonar la configuración ORM del servidor y aplicarla sobre la base de datos *PostgreSQL*

11. Preparar `steam-scraper` como una tarea `cron` [50] que se lanzará diariamente

12. Ejecutar `steam-scraper` para generar una base de datos inicial

13. Lanzar el servidor escuchando en el puerto `80`

Con estas instrucciones, se genera la imagen de *Docker* que se ejecutará en el servidor. Hasta el paso 9, se genera todo en la maquina local y solo los pasos 10 - 13 se ejecutan cada vez que se lanza la imagen en el servidor. Esto hace que las tareas completamente repetitivas solo se realicen una vez.

Una vez la imagen esta creada de forma local, se etiqueta con una versión, y se sube de forma gratuita a la plataforma *Docker Hub* para que sea accesible desde todo el mundo.

Para preparar la infraestructura del servidor se ha utilizado otro fichero con más instrucciones nombrado `docker-compose.yml` y que se ha mencionado previamente en el capítulo 7.2.

8.1.2. Preparación de infraestructura con Docker Compose

`Docker Compose` es una herramienta que permite orquestar distintos contenedores Docker entre ellos y de forma exterior. Así como la configuración del `Dockerfile` prepara la imagen del servidor, `Docker Compose` señala la estructura en la que se organizan los contenedores del servidor y la base de datos, la forma en la que estos se comunican entre sí, las distintas variables de entorno requeridas para que el servidor se ejecute de forma correcta, y la forma en la que se exponen los distintos puertos de toda la solución de cada al público.

Una vez el fichero esta listo, todo lo que necesita hacer aquel que desee poner en marcha un servidor es obtener el fichero `docker-compose.yml` y ejecutar `docker compose up` en el directorio del fichero. Esto descargará las imágenes necesarias, lanzará contenedores basados

en dichas imágenes y ejecutará el servidor y el servicio de base de datos, funcionando a través del puerto `80`.

Despliegue en la plataforma de Google Cloud

Google Cloud es la única plataforma de servicios en la nube con la que el personal de desarrollo tiene alguna experiencia y por eso ha sido escogida a pesar de que otras plataformas como *Microsoft Azure* o *Amazon Web Services* ofrezcan soporte para servicios basados en contenedores *Docker*.

Para el lanzamiento de una versión de prueba durante la fase de pruebas en *Alpha* se ha procedido a generar una maquina virtual con la menor cantidad de recursos posibles, las cuales ofrecen una serie de minutos de uso gratuitos al mes, procurando así disminuir los costes de ejecución tanto como sea posible. Esta configuración incluye:

- 1 vCPU
- 0,6 GB de Memoria RAM
- 30 GB de almacenamiento persistente
- IP externa fija

Google Cloud ofrece 744 horas de uso gratuito de este tipo de instancias, que aunque son muy limitadas, suficientes para un pequeño círculo de usuarios durante la fase de *Alpha testing*. Un problema con este tipo de instancias es que todo el trabajo realizado en la escalabilidad de las distintas tareas entre múltiples hilos queda completamente anulado, ya que dicha maquina virtual solo contiene un procesador virtual con un único hilo de ejecución, sin embargo, en el caso de necesitar o querer desplegar el servidor en una maquina virtual que contenga un mayor número de hilos de procesamiento en paralelo, solo sería necesario cambiar las especificaciones de la maquina desde los ajustes de *Google* sin necesidad de alterar una sola línea de código en el servidor en los ficheros de configuración de *Docker* o *Docker Compose*.

Aunque *Google Cloud* contiene si propia solución para el despliegue de contenedores a través de *Kubernetes*, debido a que la migración de *Docker* a *Kubernetes* es un proceso por el que no es necesario pasar, se ha optado por lanzar esta máquina virtual vacía y lanzar *Docker* dentro de ella.

Figura 8.1: *Acceso por SSH a la maquina virtual en Google Cloud*

Para ello tan solo es necesario conectarse a la maquina virtual a través de SSH utilizando el botón que se puede ver en la figura 8.1, colocar el fichero `docker-compose.yml` dentro de esta y lanza el comando:

```
docker run --rm \
-v /var/run/docker.sock:/var/run/docker.sock \
-v "$PWD:$PWD" \
-w="$PWD" \
docker/compose:1.24.0 up
```

Tras lanzar este comando el proceso del servidor comenzará, mostrando una salida de texto similar a la de la figura 8.2.

Una vez configurada la máquina virtual, solo es necesario cambiar el proveedor de servicios web de la aplicación móvil para que esta apunte a la IP externa de la máquina virtual.

8.2. Retroalimentación

Tras dos semanas de uso, la retroalimentación recibida destaca los siguientes puntos a tener en cuenta:

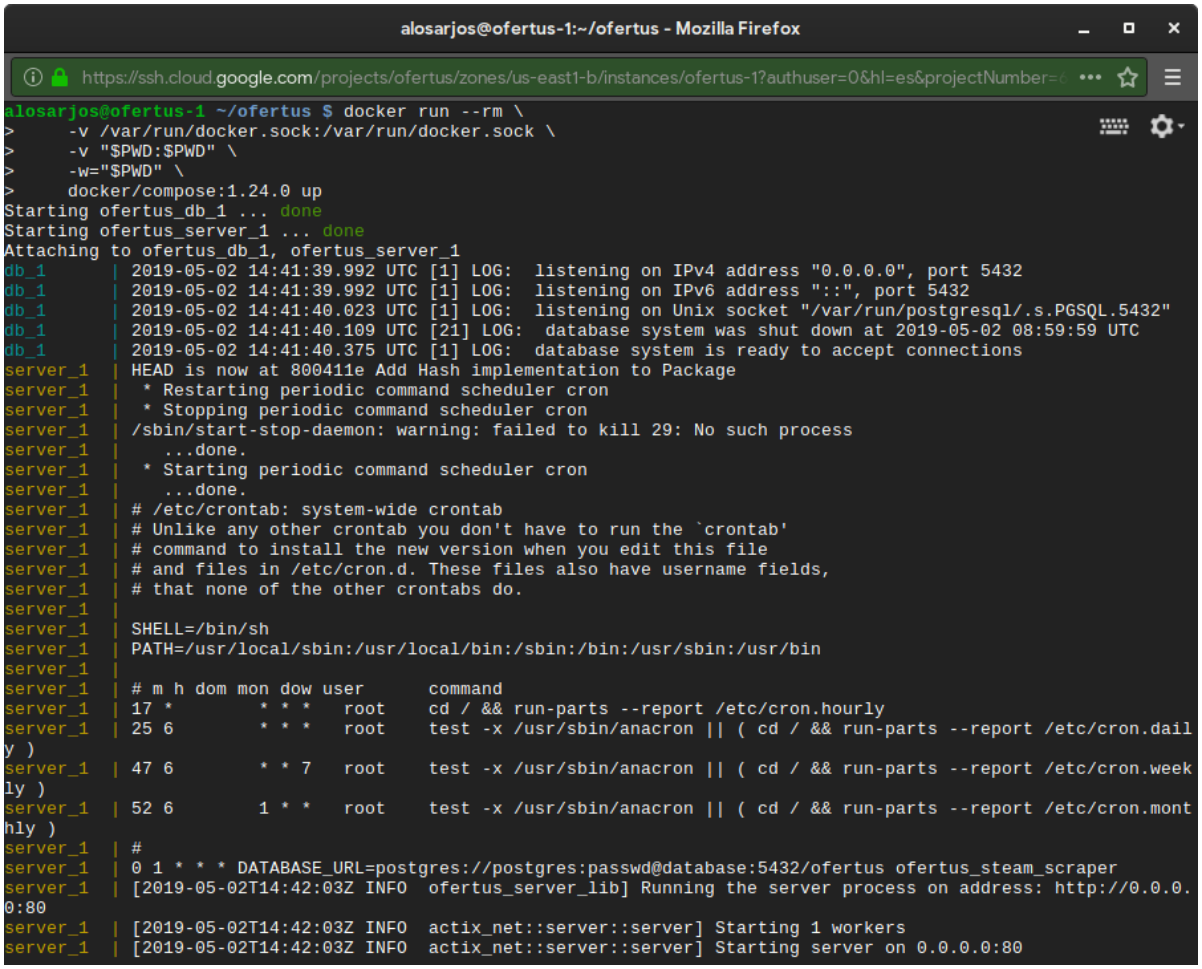

Figura 8.2: *Proceso del servidor ejecutándose en la maquina virtual en Google Cloud*

8.2.1. Flujo de trabajo

Hay usuarios que no están de acuerdo con el flujo de trabajo diseñado para la aplicación. El flujo actual de trabajo se traslada a la búsqueda de paquetes para ser añadidos a la lista de favoritos y posteriormente acceder a los detalles de estos, mientras que ciertos usuarios consideran que un mejor flujo de trabajo consistiría en acceder directamente a la vista de detalles de un paquete desde el buscador y desde la vista de detalles añadir el paquete si se desea a la lista de favoritos. Se comparan el flujo de trabajo actual y el deseado en la figura 8.3.

Figura 8.3: *Comparativa entre flujo de trabajo actual y el deseado*

8.2.2. Compras directas desde la aplicación

Algunos usuarios desearían poder adquirir paquetes directamente desde la aplicación en vez de que esta abra la página web del sitio que vende dicho paquete. Esto implica problemas importantes como la necesidad de registrarse en la web de compra, el acceder a la clave de activación una vez el proceso de compra se ha realizado, y mantener unos altos estándares de seguridad para compras y pagos *online* dentro de la aplicación. Es por ello que esta idea queda completamente descartada por el momento.

8.2.3. Mejorar transiciones entre vistas

La transición entre algunas vistas como la de carga de base de datos y la lista de paquetes favoritos no es tan fluida como se desearía. Solucionar este problema implica alterar las vistas de dichos elementos para que compartan una animación con transición en el momento en el que se realiza el cambio. El equipo de trabajo está de acuerdo con este cambio y se solucionará una vez este trabajo haya sido entregado y presentado.

Capítulo 9

Conclusiones y desafíos

9.1. Conclusiones

En este trabajo de fin de máster se ha desarrollado un servidor y las múltiples bibliotecas que este requiere para la recogida y procesado de información relativa a los precios de videojuegos en múltiples tiendas digitales. Se ha realizado una aplicación capaz de crear una base de datos en base a los videojuegos disponibles en la plataforma *Steam* así como una aplicación móvil que consume los datos ofrecidos por el servidor y los muestra en una interfaz funcional y cuidada. Para ello se han utilizado las últimas tecnologías disponibles, empleando *Rust*, uno de los lenguajes de programación con menos tiempo en el mercado pero que gana simpatizantes a un ritmo mucho mayor que otras alternativas y *Flutter*, que permite la creación de aplicaciones nativas para dispositivos móviles con distintos sistemas operativos utilizando el mismo código. También se ha desplegado el sistema en una plataforma real (*Google Cloud Platform*) a través de una de las herramientas con mayor protagonismo en los últimos años, *Docker* y se ha puesto a disposición de un conjunto de testers una versión *alpha* con la que se ha obtenido resultados muy positivos y retroalimentación de gran valor.

El objetivo principal tal y como se describe en la sección 1.2 era el de realizar una solución profesional que permita acceder fácilmente a los precios de los videojuegos en distintas tiendas digitales y aunque por el momento Ofertus no cuenta con un gran número de tiendas, el grupo de *testers* y el equipo de trabajo lleva utilizando la aplicación durante semanas, a

diferencia de otros proyectos pequeños donde la aplicación ha dejado de utilizarse al poco tiempo, esta aplicación ya es útil en un entorno real, y por ello se considera que se ha cumplido con el objetivo de ofrecer una solución real y profesional, aunque hay pequeños detalles donde se puede mejorar.

Los objetivos secundarios (mencionados previamente en la sección 1.3) eran conocer y aprender a utilizar *Rust* y *Flutter*. Durante el desarrollo del servidor, se han empleado todo tipo de elementos de *Rust*, tales como las estructuras `Option` y `Result` que garantizan una mayor estabilidad del proceso en tiempo de compilación, se han utilizado múltiples bibliotecas de terceros y se han implementado soluciones para el *web scraping* de *Steam* y en el proceso del servidor que funcionan de forma paralela haciendo uso de los múltiples hilos de ejecución que pueda tener la máquina anfitrión, logrando así un rendimiento óptimo. Por otro lado se ha realizado una aplicación con *Flutter*, que ha requerido aprender a trabajar con lógica asíncrona asociada al dibujado de una interfaz síncrona. Se ha hecho uso de material aprendido durante el máster y se ha obtenido una aplicación móvil que responde rápidamente a las interacciones del usuario y se mantiene en un nivel de rendimiento aceptable durante su ejecución. Además se ha realizado el trabajo de internacionalización y se han realizado los cambios oportunos para asegurar su compatibilidad con el tema oscuro que ofrecerá *Android Q* en un futuro, por lo que se considera una solución que hace uso de las últimas novedades y se mantiene en un alto grado de estabilidad, obteniendo por lo tanto el grado de profesionalidad que se esperaba.

9.1.1. Rust

Aprender *Rust* ha sido una oportunidad para volver a los lenguajes de sistemas, donde uno tiene que volver a responsabilizarse del uso de la memoria, donde se debe controlar a bajo nivel la ejecución de procesos en paralelo pero donde el rendimiento se vuelve una clave fundamental y esto queda demostrado en las pruebas de rendimiento que se han realizado sobre el proyecto. *Rust* es un lenguaje de programación con una base y unos principios muy

bien trabajados, con una curva de aprendizaje ideal para los recién llegados, una documentación mucho más accesible que la de otras alternativas como C++ y más seguridad que alternativas como Python, con una comunidad que apuesta muy fuerte por su futuro. Es un lenguaje con el que quiero seguir trabajando y aprendiendo definitivamente.

9.1.2. Flutter

Respecto a *Flutter*, ha sido una maravilla y a la vez un poco decepcionante en lo relacionado a la multiplataforma. Si bien es cierto que *Flutter* ofrece hasta cierto punto el concepto de "Un código base, múltiples plataformas", en el código se debe especificar entre los *widgets* de *iOS* o *Android*, haciendo que una de las plataformas no tenga los elementos visuales característicos de las aplicaciones nativas de esta.

Es cierto que, si el equipo de desarrollo cuenta con una sección de diseño, es posible generar una aplicación con un lenguaje de diseño propio y crear todos los *widgets* de forma personalizada ofreciendo una aplicación que a pesar de no contar con los elementos propios de la plataforma, funcionará con un muy buen rendimiento, pero en este caso en el que el equipo de trabajo se ha limitado a un programador, el resultado final es que la aplicación, en lo relacionado a su aspecto visual, no se encuentra en el estado que me gustaría para la plataforma de iOS.

Sin embargo debo destacar que se trata de un marco de trabajo realmente fácil de aprender y de acceder. El rendimiento es óptimo, las funcionalidades como la recarga en caliente son realmente útiles y está perfectamente preparado para integrarse con los editores de código más populares a día de hoy. Si ofrecieran algún tipo de *widget* que en tiempo de compilación se adaptara a los ofrecidos de forma nativa por *iOS* o *Android*, se convertiría muy probablemente en el marco de trabajo más popular para aplicaciones móviles.

9.1.3. Docker

Docker es una herramienta magnífica para trabajar de manera agnóstica a la plataforma utilizando servicios que deben ejecutarse en segundo plano como ha sido el caso de un

sistema de bases de datos.

Permite tener un control total sobre los distintos procesos sin que estos lleguen a afectar a la máquina anfitrión y mantiene el sistema operativo nativo aislado de todos los ficheros y operaciones que requiera haciendo que el mantenimiento o la eliminación de estos servicios sea mucho más sencillos de realizar que de forma nativa.

A pesar de ofrecer una capa de abstracción que en general facilita mucho las operaciones sobre el contenedor, la pérdida de rendimiento ha resultado inapreciable y tras usarlo durante varios meses, es fácil ver por qué se ha convertido en una de las herramientas clave en la evolución de los sistemas en la nube.

9.1.4. Proyecto basado en Web-Scraping

Es una lástima que ni las tiendas online ni el propio *Steam* ofrezcan una *API REST* a la que conectarse y ello implica que las bases sobre las que se ha creado este proyecto sean realmente frágiles. En el momento de escribir esta documentación, temo que *Steam* pueda rediseñar completamente su web rompiendo por completo el código creado hasta ahora sobre el que se basa la base principal de este proyecto.

Ha habido momentos durante el desarrollo del proyecto en el que se han encontrado problemas que requerían toda la atención una vez aparecían, como por ejemplo el caso de las precompras en las tiendas online donde se buscan los precios. Un caso de uso de los múltiples que pueden aparecer en una web sin previo aviso y que requirió atención urgentemente durante el desarrollo ya que no era posible saber cuándo habría otro caso de precompra contra el que probar el código.

Trabajar con *web-scrapping* al final del día implica que el proyecto depende completamente de lo que el equipo de desarrollo de dicha web decida hacer y puede que incluso en algún momento, el mantenimiento de un módulo de *web-scraping* para un sitio web sea inviable.

9.1.5. Proyecto en general

Crear y mantener un proyecto a nivel profesional no es fácil aún tratándose de un concepto tan pequeño como puede ser este. El trabajo relacionado con el *web-scraping* puede cambiar en cualquier momento y requerir de mantenimiento completo para volver a funcionar, pero el servidor está preparado para aceptar que el búscador de precios de una web falle simplemente no mostrando el precio en ese sitio como si el juego no estuviera disponible ahí.

La aplicación todavía requiere de pequeños retoques visuales, como animaciones, pequeñas modificaciones en márgenes y tamaños y transiciones para sentirse completamente profesional, pero son estos detalles los que, sin ayuda de un encargado del diseño, toman más tiempos del disponible.

El uso de *logs* y el diseño de un servidor que contemple los casos de uso y los diferentes errores que puedan ocurrir durante su utilización también es un tema que se ha vuelto recurrente durante las iteraciones a lo largo de su implementación. A modo de prueba se refactorizó el código para hacer que tanto el servidor como la herramienta de *web-scrapping* que añaden la información a la base de datos se pudieran utilizar como herramientas de *CLI* con sus correspondientes parámetros a la hora de utilizarlos. Tan solo con la cantidad de código, diseño y manejo de errores que supuso esta pequeña tarea, realizar la comprobación de errores en la misma proporción para todo el resto del funcionamiento del servidor hubiera supuesto también la necesidad de una cantidad de tiempo no disponible, pero es una de las tareas que personalmente más me interesan porque considero el manejo de errores como algo fundamental cuando se trata de desarrollar software robusto y confiable.

9.2. Ideas descartadas

Durante la fase de desarrollo se han tenido en cuenta distintas ideas y soluciones para ciertos problemas según se han ido apareciendo. Pero hay ciertas ideas que se han tenido que descartar debido a la falta de tiempo y a la relación de tiempo requerido y mejora en los resultados que ofrecían.

9.2.1. Recortador de imágenes inteligente

Las imágenes y las distintas proporciones respecto a la relación de aspecto de estas fueron un problema siempre. Las imágenes no quedan perfectas y por lo tanto la aplicación pierde parte del componente profesional que se busca. Mi primera opción fue siempre la de simplemente ajustar la imagen a la altura del espacio proporcionado al elemento, sin embargo hubiera preferido recortar la imagen horizontalmente de forma inteligente para que resultará en una experiencia visual que encajara mejor con la aplicación.

Tras realizar varias pruebas, se pudo comprobar que no era viable por la obtención de resultados impredecibles, especialmente en imágenes que se aprovechan completamente, haciendo así que los recortes resultaran en deformaciones de la imagen.

9.2.2. HashMaps concurrentes

Para la generación de la base de datos, se generan dos *HashMaps* en memoria que al serializar en ficheros, quedan convertidos en ficheros con formato *JSON*. Originalmente las numerosas entradas de paquetes disponibles en *Steam* hizo pensar que se requeriría de un *HashMap* al que poder acceder concurrentemente por parte de cada hilo encargo del *web-scraping* para poder realizar esta tarea rápidamente. Tras realizar varias pruebas, se llegó a la conclusión de que el acceso al *HashMap* para la inserción de datos no llegaba a ser un cuello de botella durante el proceso y que éste, de forma normal se realizaba en un tiempo aceptable.

El haber implementado y utilizado *HashMaps* concurrentes hubiera supuesto una alta inversión de tiempo y una complejidad añadida al proyecto para una mejora que tan solo hubiera supuesto de unos segundos.

9.3. Desafíos

No hay muchas alternativas a esta aplicación móvil en el mercado, tan solo en la Web, donde la capacidad de cómputo es mucho mayor. Sin embargo los mayores obstáculos se han

podido superar para obtener así una solución a la altura de las expectativas.

9.3.1. Obtener una base de datos de videojuegos

No existe una base de datos de videojuegos al completo. Por lo tanto había que analizar la situación actual respecto a la fuente de videojuegos existente para que el usuario pueda realizar búsquedas locales.

Steam no ofrece ningún tipo de *API* a la que poder acceder adecuadamente. Sí que ofrece una *API* con un listado con un buen número de elementos que contiene en su tienda pero no están organizados, el listado es incompleto y agruparlo en las necesarias categorías era inviable.

Realizar *web-Scrapping* era la forma más insostenible de realizar este proyecto pero la única tras comprobar que *SteamDB* también carece de acceso a través de una *API*. Y cuando se trata de realizar *web-Scraping* a cerca de 54000 elementos donde cada uno puede contener variantes diferentes en la forma de mostrar los precios, la tarea se vuelve realmente complicada. No solo se trata de una tarea compleja de realizar basada en algoritmos que pueden dejar de funcionar en cualquier momento, también es una tarea que requiere una gran cantidad de tiempo de computación para realizarse. Durante la fase original de desarrollo se implementó un *scrapper* basado en la *API REST* previamente mencionada en *Python*. Debido a los límites de *Steam* en lo que respecta a número de peticiones por tiempo hubo que limitar el programa, pero el resultado final fue un proceso que conllevaba cerca de tres días en ejecutarse y los resultados quedaban lejos de ser tan precisos como se esperaba.

9.3.2. Crear un motor de búsquedas rápido y con sugerencias en tiempo real

Aun sin experiencia previa realizando un motor de búsqueda, los conceptos aprendidos durante la asignatura de "sistemas de gestión de datos y de la información", sirvieron de guía para empezar a buscar información sobre cómo crear un sistema de recuperación de la información que se ajustará a las especificaciones del proyecto. Finalmente se optó por un

motor de búsquedas *booleano* basado en los índices invertidos vistos durante el máster junto al uso de *tries* para la generación de palabras que utilizar como claves de búsqueda según el usuario escribe.

9.3.3. Manejo de errores

Uno de los inconvenientes del proyecto es que la parte de *web-Scraping* ya de por sí resulta muy volátil, pero también hay que tener en cuenta la posibilidad de que una tienda no disponga de algún juego. La cantidad de errores y posibles referencias y comprobaciones contra elementos nulos era muy alta en el proyecto. Por esta entre otras razones se optó por el uso de *Rust* al ser un lenguaje de programación que fuerza en tiempo de compilación a verificar las distintas posibilidades que puede tener una operación donde el resultado no es seguro.

9.3.4. Facilidad a la hora de crear e insertar Pricers

Aunque no hay experiencia previa recibiendo colaboraciones externas en proyectos, estas siempre son bienvenidas. Crear y mantener un *pricer* puede ser una tarea complicada, crear y mantener varios definitivamente lo es, por ello se decidió diseñar el sistema de *pricers* de la manera más sencilla posible para que su inserción y ejecución en paralelo fuera simple y a su vez fuera sencilla la implementación de uno nuevo por parte de desarrolladores externos.

Capítulo 10

Trabajo futuro

El desarrollo de la plataforma no termina con la versión actual. Hay múltiples funcionalidades planeadas para ser implementadas en el futuro que no se han podido añadir ahora por falta de tiempo o recursos.

10.1. Notificaciones en tiempo real

Una de las ideas originales del proyecto es la de obtener notificaciones en tiempo real sobre el estado de los precios de los paquetes bajo seguimiento. Durante el desarrollo de la aplicación se trabajó en la implementación de esta funcionalidad, sin embargo en ese momento el marco de trabajo no tenía ninguna funcionalidad nativa para este tipo de tareas, requiriendo de modificaciones a bajo nivel exclusivas para la plataforma de *Android*.

Actualmente la plataforma tiene la capacidad de enviar notificaciones *Push* a través de las herramientas de *FireBase* de *Google*. Esto implica que un administrador puede enviar notificaciones a los dispositivos deseados o que el servidor puede enviar notificaciones a estos a través de la *API REST* que *Firebase* ofrece. Esta opción se ha mantenido como una forma a través de la cual legar cierto tipo de notificaciones a los usuarios por parte de la administración, pero es el método deseado para notificar sobre ofertas relacionadas con los paquetes añadidos a la lista de favoritos.

Actualmente, sin hacer uso de ello, la aplicación incorpora la ejecución de tareas programadas en segundo plano, el cual es el primer paso para la comprobación de precios y

notificaciones sobre ofertas periódicas que se desea implementar dentro del diseño de la aplicación, sin embargo el uso de notificaciones locales dentro del dispositivo es una funcionalidad sobre la que el marco de trabajo de *Flutter* sigue trabajando a día de hoy y sobre la que se hace un seguimiento para poder implementar esta funcionalidad de manera adecuada una vez sea posible.

10.2. Optimización en la entrega de la base de datos

Aunque los ficheros JSON que contienen la información referente a la base de datos para ser posteriormente transferidos a la aplicación móvil se han mantenido dentro de un tamaño que se considera reducido, se pretende implementar un sistema de actualización diferencial entre versiones de los ficheros de las bases de datos.

La estructura de cada paquete podría incrementar en un futuro para poder añadir así más información y poder así implementar una mayor funcionalidad en la aplicación. Para mantener el uso de datos móvil bajo mínimos, un sistema que tan solo descargue un delta entre la versión existente en el dispositivo y la ofrecida por el servidor sería ideal.

La idea original trataba de hacer que el servidor mantuviera una versión *delta* respecto a la anteúltima versión y una versión completa con la última, pero este caso de uso tan solo sería aplicable a los usuarios activos que accedieran a la aplicación casi a diario y por ello se descartó. Se requiere de un sistema más complejo para poder llevar a cabo esta tarea de manera adecuada y *Dart* actualmente no ofrece un mecanismo nativo para la transmisión de datos de esta manera como podría ser *rsync*.

10.3. Envío del HTML de las tiendas desde el dispositivo móvil

Desgraciadamente muchas tiendas online no tienen la opción de obtener manualmente los precios en un tipo de precio regional específicamente y se limitan a mostrar el tipo de divisa para la región desde la que llega la petición *HTTP*. Esto causa algunos problemas en

el momento en el que el servidor se sitúa en otra región como por ejemplo Estados Unidos. La mejor solución a corto plazo sería obtener el código *HTML* correspondiente para el *web scraping* en el dispositivo móvil y enviarlo al servidor para que realize el procesado de la información. Esto aumentaría ligeramente el consumo de datos desde el móvil y añadiría algo más de latencia en la obtención de precios, pero es la mejor solución que tiene el equipo de trabajo para solventar este problema.

Capítulo 11

Introduction

11.1. Purchasing videogames on the Internet

The number of videogames sales in digital format has been growing at a high rate during the last years, with Steam increasing the number of users and active buyers year after year [2]. With the goal of avoiding the monopoly on the digital videogame market inside the PC sector, Steam allows to the publishers to distribute keys through other ways to later be activated on Steam and still get access to the same features as the user who decides to buy the game directly from Steam. This allows the user to find lots of digital stores competing among themselves with different discounts and promotions for the exact same product: a license for a videogame on Steam. Besides, different digital stores can have their games with discounts during different periods, making the purchase of an specific game on a concrete moment more profitable on one specific store. This project aims to ease the process of finding a game at its best price in a specific moment without the need of browsing through multiple websites comparing prices manually. All the project is available on `https://gitlab.com/ofertus`.

11.2. Main goals

The objective of this project is to offer to the possible game buyers the relevant information regarding the prices of different videogames in the most simple way possible.

The main goal of the project is to implement a professional rated solution where the end user will be able to search easily between the different prices for the videogames he or she wishes among the different online stores where this product is offered. On the Web there are already some similar solutions, such as IsThereAnyDeal [3] which offers different prices for a videogame among different sites. With this project, which will be called Ofertus from now on, the aim is to offer a REST API capable of offering the same information with an standardized and specified format, allowing this way the development of different applications so that third party developers can create their own solutions to make easier the searching of prices to the users.

This solutions will be implemented as the server product, but at the same time, a mobile application has been developer to consume the information provided by the server as a way to show the server capabilities to retrieve the information with the correct format about the different videogames and the packages offered by Steam on its digital store. In the case of Android, the Google Play Store has different applications to check prices, focused on specific stores or where a user driven community share the offers between themselves, but there is not a single solution that offers directly a comparison for videogame prices in a automated way between different digital stores and Ofertus could be the first solution of this kind on the mobile market.

11.3. Secondary goals

The secondary objectives of this project include the management of a project alongside a community and the learning of different tools and programming languages recently launched.

All the Ofertus project has been created with the intention of being published as open source, distributed under the MIT [4] license and with free access through the GitLab [5] platform. The goal of the project is to find new developers who wishes to help improving the functionality or adding new digital stores to get analyzed by the service. There has been a special emphasis on making the code easy to read and understand, and so on the making of

abstractions and libraries to make it easier for third party developers to collaborate adding new functionalities to the project.

On other side, Ofertus is a project centered on a relatively simple concept, this makes this project an ideal place to learn and experiment with new languages and development tools without required complex algorithms or advanced knowledge. Rust has gained the the favor of lots of developers on the context of system languages and it is now the most beloved programming language in the latest Stack Overflow [6] survey, a renown programmer community. Flutter is now positioned as a great choice on the context of mobile multiplatform solutions, with a different approach to its adversary React Native. Because Flutter is a much younger product, and because it is not based on JavaScript, it has become the most interesting mobile solution to experiment with.

11.4. Planning and work-plan

The planning of the project is based on the available time for the development, fixing May 1st as the feature freeze date. From this date on, all the work will focus on bug fixing and the documentation.

11.4.1. Task organization

Because there is only one person working in the project development, this person will be the manager on all the tasks required by the project.

11.4.2. Schedule of events

Creation of a service to provide prices from Steam

The first step will be making a basic server on Rust that will return on JSON format the basic prices information of a package in Steam. This task was done between the 21st of December of 2018 and the 28th of December of that same year.

Developing the first application prototype

Once there was access to the basic information from Steam, began the work on the first drafts and the implementation of a mobile application with Flutter. The application had to show the information provided by the server with a visually pleasant design. During this stage there was iterative work regarding the final design of the application. Later on the evolutional process regarding the design is explained. This process was done between the 29th of December of 2018 and the 16th of January of 2019.

Creating a videogame database in order to search

After developing the first prototype where the information about the packages which prices should be shown was added manually, the next step was creating a videogame database that would allow the future creation of a local search engine inside the mobile device. After trying different mechanism and tools, at the end the development team picked the actual system based on web scraping and worked on it exclusively from the 17th of January of 2019 until the 10th of February of 2019. Still, the development team keeps working on it to solve problems regarding all the different use cases that exist on the Steam site.

The search engine

Once the development team achieved a database based on the videogames offered by the digital store Steam, the work began on the creating of a local search engine for the mobile devices based on this data. This required the research of data structures in order to get a solution that could provide results in a a efficient way on a mobile device. This work was realized between the 11th of February of 2019 and the 24th of February of the same year.

THe development of different Pricers

The next step was creating different libraries for the project that the server would use to search the prices of a Steam package on different digital stores. This process required of multiple iterations and tests because of all the different situations that can happen regarding

a package on the different sits (With out without a deal, regular sale or presale, being free temporally...). For now four different pricers have been developed and still the team keeps working on them when a bug is found during the application test period. Among these pricers it was also created the interface that would allow to a third party developer to create its own web scraper and making it easy to add it to the already existing ones on the server. This task was done between the 25th of February of 2019 and the 2nd of April of that same year.

Development of the mobile application

The last step of the development phase was ending the mobile application. The development has transitioned though a couple of code refactors to keep it inside the Flutter standards that has changed slightly over the time. Platform filters for the searches, animations, transitions, a page with all the licenses used by the libraries used, a dark mode and some extra functionalities that will be explained on their respective section have been added. The development of the application has been done between the 3rd of April of 2019 and the 28th of April of 2019.

Deployment of the solution on the cloud

The last step was done between the 29th of April of 2019 and the 6th of may of the same year deploying the solution on the Google cloud. This process includes generating and orchestrating the server in form of Docker container and deploying them on the Google Cloud platform.

Testing the application and fixing bugs

Once the server was available publicly, the application was sent to a selected group of individuals to tests it and solve the different problems that could appear and also adding some minor features or making little design changes. This process was done between the 8th of may of 2019 and the 20th of may of the same year, where the project gets tagged

as beta, and the development of this is finished until the finalization of the corresponding documentation.

Capítulo 12

Conclusions and challenges

12.1. Conclusions

During this master's thesis a server and multiple libraries have been developed to retrieve and process the information relative to the pricing of videogames on multiple digital stores. We have created an application capable of creating a database based on the packages available on the Steam platform and also a mobile application that consumes the data offered by the server and shows it in a functional and polished interface. In order to achieve this, the latest technologies have been used, using Rust, one of the languages with less time in the market but gaining fans faster than other alternatives and Flutter, that allows creating native applications for mobile devices with different operation systems using the same code. The system has also been deployed on a real platform (Google Cloud Platform) through one of the leading tools during these last years, Docker and an alpha version of the application became available for a selected group of testers, getting very positive results and valuable feedback.

The main objective as it has been described on the section 11.2 was to create a professional solution that allows accessing easily to the prices of videogames on different digital stores, and while for the moment Ofertus does not count with a large number of stores, the group of testers and the development team have been using the application for weeks. In contrast to other small projects where applications have been stopped being used in a

short period of time, this application is already useful on a real environment, and because of this, it is considered that the objective of offering a real and professional solution has been achieved, although there are still some minor details that can be improved.

The secondary objectives (previously mentions on the Section 11.3) were to learn and understand how to use Rust and Flutter. During the development of the server, all kind of Rust elements have been used, such as `Option` and `Result` structs, that guarantees better stability of the process during compile time, multiple third party libraries have been used and solutions on the web scraping of Steam and the server have been developed that works on parallel making use of the multiple threads available on the host machine, achieving this way optimal performance. On the other side an application with Flutter has been developed, which has required to learn how to work with asynchronous logic related with synchronous rendering of the user interface. Materials learned during the master's degree has been applied and a mobile application that responds quickly to the user interactions and keeps its performance on acceptable levels has been obtained. Also there has been internationalization work on the applications and dark theme compatibility has been ensured for the future Android Q release, and so, it is considered a solution that makes use of the latest updates and keeps a great level of stability, obtaining the professional quality that was expected.

12.1.1. Rust

Learning Rust Rust has been a opportunity to go back into system languages, where the developer has to take responsibility of the memory usage, where the execution of parallel code has to be controlled, but also where the performance becomes key and this has been shown on the performance test done on this project. Rust is a programming language with a really well elaborated base and principles, with an ideal learning curve for newcomers, a more accessible documentation than other alternatives such as C++ y with better security than other alternatives like Python, with a community that bets strongly on its future. It

is a language I want to keep working on and learning definitely.

12.1.2. Flutter

Regarding Flutter, it has been a wonder and at the same time a bit of a disappointment regarding multi-platform compatibility. If it is true that Flutter offers up to some point the concept of "A single base code, multiple platforms", on the code must be specified if the widgets will be iOS or Android based, making then one of the platforms no getting the visual elements featured on the native applications.

It is true that, is the development team has a design department, is possible to create an applications with its own design language and so, make all the widgets on a customized way, offering an app that, even if it does not have the platform own visual elements, will still get good performance, but because in this case the development team is limited to a single programmer, the final results is that the application, regarding the visual aspect, it is not on the desirable state regarding the iOS platform.

However I should mention that it is a really easy framework to learn and to start with. Performance is optimal, functionalities such as the hot reload are really useful and it is completely ready to be integrated with the most popular code editors available nowadays. If they would offer some kind of widget that would change into the native solutions offered by iOS or Android on compile time, it would probably become the most popular framework for mobile applications.

12.1.3. Docker

Docker is a magnificent tool to work in an agnostic way to the platform using services that must be executed in background like the case of the database system.

It allows to have the complete control over the different processes without allowing them to affect the host machine and keeps the native operating system isolated from the files and the operations that it may require, making the maintenance or the removal of these services much easier to do compared to the native installations.

Even if it offers an abstraction later which generally makes easier the operations over the container, the performance loss has been inappreciable and after using it for some months, is easy to see why it has become in one of the key tools during the evolution to the cloud systems.

12.1.4. Project based on Web-Scraping

It is a pity that either the online stores or Steam do not offer an API REST to connect and use, and that means that the basis that have been used for this project are really fragile. In the moment of writing this documentation, I fear that Steam may change completely the design of their site, breaking completely the code created until now where most of this project is based on.

There have been some moments during the development of the project where some problems would appear that required the development team complete attention once they showed up, like for example, the case of the prepurchases on digital stores where prices are searched. Un single case of all the possible ones may appear without any previous warning y it required urgent attention because it was not possible to know when a new case of prepurchase could appear in order to test the fix.

Working with web-scraping at the end of the day means that the project depends completely on whatever the development team of the different sites decide to do and maybe someday, the maintenance of a module for web-scraping for a site may not be possible.

12.1.5. Project in general

Creating and maintaining a project at a professional level is not easy even with dealing with such a small concept as this one. The work related with the web-scraping may change at any moment y require complete maintenance in order to be functional again, bu the server is ready to accept the fact that a price searcher of a site may fail simply not showing the price for that site like if that concrete package was not available on that store.

The mobile application still requires of small visual tweaks, like the animations, small

changes on margins, sizes and transitions to feel completely professional, but this kind of details, without the help of a designer, take more time that the available one.

The usage of logs y the design of a server that can handle the different use cases and errors that may happen during its utilization has also been a recurring theme during the different iterations while implementing it. As a test, the code regarding the server and the web-scrapping tool were refactored so they could be used as CLI tools with their respective parameters when using them. Just with the amount of code, design and error control that was required to do this little task, doing error handling at the same level for the whole server would have required an unavailable amount of time, but it is still one of the most personally interesting tasks, since error handling is essential to develop robust and reliable code.

12.2. Discarded ideas

During the development phase, the team has considered different ideas and solutions for some problems that have showed up. But there have been some ideas that had to be discarded because of the lack of time or because the benefits of implemented them were not worth the time.

12.2.1. Smart image crop

The pictures and their aspect ratios have been always a problem. The images cannot show up perfectly fitting and this makes the application loose the professional look it is trying to accomplish. The first option was always fitting the image to the height of the available space, but the team would have preferred to crop the image in a smart way to achieve a better visual experience.

After making some tests, the team checked that implementing this feature was not viable because of the unpredictable results obtained, specially with the pictures that make complete usage of their canvas, obtaining deformed or badly cropped pictures.

12.2.2. Concurrent HashMaps

For the database generation, two HashMaps are created in memory that, once they are serialized into file,s get converted into JSON files. At first glance, the amount of packages available on Steam made the team think about requiring the usage of a HashMap that should be able to get accessed concurrently by each thread in charge of web-scraping in order to get this task done as fast as possible. After some tests, the results where that the access to the regular HashMap to insert the data was not bottle-necking during the process, and it achieved its goal inside an acceptable amount of time.

Implementing and using concurrent HashMaps would have required a high amount of time and would have added more complexity to the project for a improvement of just a few seconds.

12.3. Challenges

There are not many alternatives to this mobile application in the market, only on the Web, where the computing capacity is way better. However the biggest obstacles have been overcomed in order to achieve a solution that lives up to expectations.

12.3.1. Obtaining a videogame database

There is not available anything such as complete videogame database. And because of that it was required to examine the actual situation regarding the source of existing videogames in order to allow the user to do searches locally.

Steam does not offer any kind of API to access the information. It offers an API with a list of elements inside its store but it is unorganized, the list is incomplete and grouping it into categories wasn't viable.

Doing web-scrapping was the most unmaintainable way of realizing this project, but it was the only way after checking that SteamDB also lacks of any API access. And when it comes to web-scrap around 54000 elements where each one can have different ways of

showing prices, the task becomes really complex. It is not simply a difficult task to do based on algorithms that can stop working at any moment, it is also a task that required a high amount of computation time to be done. During the original development a scrapper based on the previously mentioned API was done using Python. But because of the limits of Steam regarding the number of requests per time the program had to be limited. In the end, the scraper needed around three days to finish and the results were not as precise as expected.

12.3.2. Creating a fast search engine with suggestions in real time

Even without any previous experience making a search engine, the concepts learned during the course of "data and information management systems" were helpful as a guide to start search for information about how to create a information recovery system adjusted to the project specifications. Finally the selected engine was a Boolean search engine based on the inverted indexes that were explained during the master degree among the use of tries for the generation of words to use as key of the search when the user starts typing.

12.3.3. Error handling

One of the inconveniences of the project is the web-scraping, which already by itself is volatile, but the project also required to take into account he possibility of a store not having available a certain game. The quantity of errors and possible references and checks against null elements was really high in the project. Because of these among other reasons the team opted to use Rust, since it is a programming language that forces on compile time to verify all the possibilities that may have an operation where the result is not certain.

12.3.4. Creating and adding pricers easily

Even without any kind of previous experience receiving external collaboration in projects, these are always welcome. Creating and maintaining a pricer can be a complicated task, creating and maintaining some definitely is. Because if this it was decided to design a pricer

system so the addition and execution of them would be as easy as possible but at the same time making simple the implementation of a new one by third party developers.

Bibliografía

[1] Josh Pfosi, Riley Wood, and Henry Zhou. A comparison of concurrency in Rust and C.

[2] Valve. Charla y diapositivas del GDC 2019 (documentación de steamworks). https://partner.steamgames.com/doc/resources/gdc2019, abril 2019.

[3] Tomáš Fedor. Deals - isthereanydeal. https://isthereanydeal.com/, junio 2011.

[4] The MIT License | Open Source Initiative. https://opensource.org/licenses/MIT.

[5] GitLab. https://gitlab.com/, octubre 2011.

[6] Stack Overflow Developer Survey 2018. https://insights.stackoverflow.com/survey/2018, marzo 2018.

[7] Rust programming language. https://www.rust-lang.org/, mayo 2015.

[8] Apache License, Version 2.0 | Open Source Initiative. https://opensource.org/licenses/Apache-2.0.

[9] Matt Brubeck. Oxidation - mozillawiki. https://wiki.mozilla.org/Oxidation, diciembre 2017.

[10] Chris Peterson. Proyecto stylo. https://wiki.mozilla.org/Quantum/Stylo, marzo 2017.

[11] Google. Flutter - Beautiful native apps in record time. https://flutter.dev/, febrero 2018.

[12] Facebook. React Native · A framework for building native apps using react. https://facebook.github.io/react-native/, febrero 2015.

[13] Google. Dart programming language | dart. https://www.dart.dev/, noviembre 2013.

[14] Docker Inc. Enterprise Application Container Platform | Docker. https://www.docker.com/, marzo 2013.

[15] The PostgreSQL Global Development Group. PostgreSQL: The world's most advanced open source database. https://www.postgresql.org/, enero 1997.

[16] The PostgreSQL Global Development Group. Postgresql: About. https://www.postgresql.org/about/.

[17] The PostgreSQL Licence | Open Source Initiative. https://opensource.org/licenses/postgresql.

[18] Nikolay Kim. Actix - Actor System and Web Framework for Rust. https://actix.rs/, agosto 2017.

[19] Sean Griffin. Diesel. http://diesel.rs/, agosto 2015.

[20] Inc. Functional Software. Sentry | Error Tracking Software — Javascript, Python, PHP, Ruby, more. https://sentry.io, 2010.

[21] Addison Wesley. Inverted Indexes. https://algo2.iti.kit.edu/download/ti_ss13_lec3.pdf.

[22] Maha Maabar. Trie Data Structure | bioinformatics i/o. http://bioinformatics.cvr.ac.uk/blog/trie-data-structure/.

[23] Blake. Diggory. rustup.rs - The Rust toolchain installer. https://rustup.rs/, septiembre 2015.

[24] Introduction - The Cargo Book. https://doc.rust-lang.org/cargo/, mayo 2014.

[25] Inc. npm. npm | build amazing things. https://www.npmjs.com/, 2014.

[26] Levente Polyak. Arch Linux - postgresql-libs (x86_64). https://www.archlinux.org/packages/extra/x86_64/postgresql-libs/, abril 2008.

[27] Google. Download Android Studio and SDK tools. https://developer.android.com/studio, mayo 2013.

[28] Aaron Griffin. Arch Linux. https://www.archlinux.org/, marzo 2002.

[29] Canonical Ltd. The leading operating system for PCs, IoT devices, servers and the cloud | Ubuntu. https://www.ubuntu.com/, octubre 2004.

[30] Microsoft. Visual Studio Code - Code Editing. https://code.visualstudio.com/, abril 2015.

[31] Danny Tuppeny. Flutter - Visual Studio Marketplace. https://marketplace.visualstudio.com/items?itemName=Dart-Code.flutter, abril 2018.

[32] Danny Tuppeny. Dart Lang - Visual Studio Marketplace. https://marketplace.visualstudio.com/items?itemName=Dart-Code.dart-code, julio 2016.

[33] Rust-Lang programming team. Rust (rls) - Visual Studio Marketplace. https://marketplace.visualstudio.com/items?itemName=rust-lang.rust, julio 2016.

[34] Rust-Lang programming team. rust-lang/rls: Repository for the Rust Language Server (aka RLS). https://github.com/rust-lang/rls, marzo 2017.

[35] Focus Multimedia Ltd. Fanatical | Buy PC Games, Steam Keys, Bundles (formerly Bundle Stars). https://www.fanatical.com/en/, noviembre 2017.

[36] Pavel Djundik and Martin Benjamins. Home · SteamDB · Steam Database. https://steamdb.info/, 2012.

[37] Brice Lambson. Migraciones - EF Core | Microsoft Docs. https://docs.microsoft.com/es-es/ef/core/managing-schemas/migrations/, octubre 2018.

[38] Rene Floor. cached_network_image | flutter package. https://pub.dev/packages/cached_network_image, mayo 2019.

[39] 2Dimensions Team. nima | flutter package. https://pub.dev/packages/nima, abril 2019.

[40] Alexandre Roux. sqflite | Flutter Package. https://pub.dev/packages/sqflite, marzo 2019.

[41] Flutter Team. share | Flutter Package. https://pub.dev/packages/share, marzo 2019.

[42] Flutter Team. shared_preferences | Flutter Package. https://pub.dev/packages/shared_preferences, mayo 2019.

[43] Flutter Team. url_launcher | Flutter Package. https://pub.dev/packages/url_launcher, marzo 2019.

[44] Dart Team. intl | Flutter Package. https://pub.dev/packages/intl, marzo 2019.

[45] Buck DeFore. ProtonDB | Gaming reports for Linux using Proton and Steam Play. https://www.protondb.com/, agosto 2018.

[46] Valve Corporation. Proton. https://github.com/ValveSoftware/Proton, agosto 2018.

[47] Searchdelegate should respect the theme more · issue #19734 · flutter/flutter. https://github.com/flutter/flutter/issues/19734.

[48] Manolis Terrovitis, Spyros Passas, Panos Vassiliadis, and Timos Sellis. A combination of trie-trees and inverted files for the indexing of set-valued attributes. In *Proceedings of the 15th ACM international conference on Information and knowledge management*, pages 728–737. ACM, 2006.

[49] Google Translation Toolkit. https://translate.google.com/toolkit/.

[50] Ubuntu Developers. Cron. https://launchpad.net/ubuntu/+source/cron, abril 2012.